A STRUCTURE FOR PLANS AND BEHAVIOR

Technical Note 109

August 1975

By: Earl D. Sacerdoti
Artificial Intelligence Center

SRI Project 3805

Report Documentation Page		*Form Approved* *OMB No. 0704-0188*

Public reporting burden for the collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to a penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.

1. REPORT DATE **AUG 1975**	2. REPORT TYPE	3. DATES COVERED **00-08-1975 to 00-08-1975**
4. TITLE AND SUBTITLE **A Structure for Plans and Behavior**		5a. CONTRACT NUMBER
		5b. GRANT NUMBER
		5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)		5d. PROJECT NUMBER
		5e. TASK NUMBER
		5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **SRI International,333 Ravenswood Avenue,Menlo Park,CA,94025**		8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSOR/MONITOR'S ACRONYM(S)
		11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT **Approved for public release; distribution unlimited**
13. SUPPLEMENTARY NOTES
14. ABSTRACT
15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**		**162**	

· ABSTRACT

This report describes progress that has been made in the ability of a computer system to understand and reason about actions. A new method of representing actions within a computer memory has been developed, and this new representation, called the <u>procedural net</u>, has been employed. in developing new strategies for solving problems and monitoring the execution of the resulting solutions.

A set of running computer programs, called the NOAH (Nets Of Action Hierarchies) system, embodies the representation and strategies discussed above. Its major goal is to provide a framework for storing expertise about the actions of a particular task domain, and to impart that expertise to a human in the cooperative achievement of nontrivial tasks.

A problem is presented to NOAH as a statement that is to be made true by applying a sequence of actions in an initial state of the world. The actions are drawn from a set of actions previously defined to the system. NOAH first creates a one-step solution to the problem (essentially "Solve the given goal"). Then it progressively expands the level of detail of the solution, filling in ever more detailed actions. All the individual actions, composed into plans at differing levels of detail, are stored in a data structure called the procedural net. The system avoids imposing unnecessary constraints on the order of the actions in a plan. Thus, plans are represented as partial orderings of actions, rather than as linear sequences.

The same data structure is used to guide the human user through a task.

Since the system has planned the task at varying levels of detail, it can issue requests for action to the user at varying levels of detail, depending on his competence and understanding of the higher-level actions. If more detail should be needed than was originally planned for, or if an unexpected event causes the plan to go awry, the system can continue to plan from any point during execution.

The key ideas that are explored in the dissertation include:

(1) planning at many levels of detail,

(2) representing a plan as a partial ordering of actions with respect to time,

(3) execution monitoring and error recovery using hierarchical plans,

(4) using procedures that represent a task domain to generate declarative (frame-like) structures that represent individual actions, and

(5) using abstract actions to model iterative operators.

The major point of the report is that the structure of a plan of actions is as important for problem solving and execution monitoring as the nature of the actions themselves.

PREFACE

This Technical Note is a slightly revised version of a dissertation submitted to the Department of Computer Science and the Committee on Graduate Studies of Stanford University in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

ACKNOWLEDGMENTS

CONTENTS

ILLUSTRATIONS

x

I INTRODUCTION

A. Overview of the Report

This report describes progress that has been made in the ability of a computer system to understand and reason about actions. A new method of representing actions within a computer memory has been developed, and this new representation, called the procedural net, has been employed in developing new strategies for solving problems and monitoring the execution of the resulting solutions.

The report will begin with a functional description of the NOAH (Nets Of Action Hierarchies) system, a set of running programs that embody the representation and strategies mentioned above. Then the Computer-Based Consultant (CBC) project at SRI will be briefly described, since that is the testbed for the validity of the ideas incorporated in NOAH.

The following three chapters discuss in detail the representation, the problem solving strategy, and the execution monitoring strategy. Each of these chapters begins with a rather personal historical perspective on the development of the key ideas in the area. This is being done in the belief that an integrated story, even a biased one, is more interesting and imparts more understanding than a simple listing of relevant work. Each of these chapters ends with a brief survey of relevant current work in the area. In each of these chapters, the overall system is treated from a different point of view. The reader who wishes to obtain an immediate perspective on the current work is encouraged to read first the initial sections of Chapters II, III, and IV.

The report will conclude with a discussion of the inadequacies of the implemented system, and with suggestions for further investigation.

B. Functional Description of the NOAH System

NOAH is an integrated problem solving and execution monitoring system. Its major goal is to provide a framework for storing expertise about the actions of a particular task domain, and to impart that expertise to a human in the cooperative achievement of nontrivial tasks. Knowledge about a task domain is supplied to the system in two ways. Knowledge about the actions that may be taken in the domain is specified in procedural form. Knowledge about the particular state of the world in which a particular problem is to be solved is given to the system as a set of declarative expressions. In addition, knowledge about the ways in which individual actions and subplans can interact is embodied within the system itself.

Thus, there are three levels of knowledge that are brought to bear on any particular problem. The system holds general problem solving knowledge; the procedural specification of actions holds knowledge about the actions possible in a particular domain; and the data base of symbolic expressions contains knowledge about a particular situation within the particular domain.

A problem is presented to NOAH as a statement that is to be made true by applying a sequence of actions in an initial state of the world. The actions are drawn from a set of actions previously defined to the system. NOAH first creates a one-step solution to the problem (essentially "Solve the given goal"). Then it progressively expands the level of detail of the solution, filling in ever more

detailed actions. The system does not need to carry this process out until it reaches the primitive actions of the system. Depending on the purpose of the plan, the planning phase may be terminated at a relatively broad-brush level of detail. All the individual actions, composed into plans at differing levels of detail, are stored in a data structure called the procedural net. This same data structure is then used to guide the human user through a task. Since the system has planned the task at varying levels of detail, it can issue requests for action to the user at varying levels of detail, depending on his competence and understanding of the higher level actions. If more detail should be needed than was originally planned for, or if an unexpected event causes the plan to go awry, the system can continue to plan from any point during execution.

C. Overview of the Computer-Based Consultant Project

NOAH was not developed exclusively for its theoretical interest, although this is emphasized in the present report. It was intended to serve a useful purpose within the context of a larger computer system, called the Computer-Based Consultant [16].

The goal of the CBC project is to produce a computer system that can fill the role of an expert in the cooperative execution of complex tasks with a relatively inexperienced human apprentice. The system will use rich channels of communication, including natural language and eventually speech. The main function of the consultant will be to aid the apprentice in the diagnosis of faulty electromechanical equipment, and the formulation of plans for the assembly, disassembly, and repair of the equipment.

The NOAH system was built to serve as the problem solving and execution monitoring component for dealing with assembly and disassembly tasks. Following is a description of the design goals for the CBC that NOAH is intended to satisfy.

The system is intended to be able to generate step-by-step plans for converting a device from one arbitrary state of assembly into another state. The system is to be able to do this at several levels of attention, from major components down to the simple fasteners.

The CBC will generate plans in a hierarchical fashion in order to interact with the apprentice at a level that matches his expertise. The system will, upon request by the apprentice, provide more detailed instructions for any step in the process.

It will answer specific questions about the instructions it has given, questions like, "What is the purpose of this step?" or "What kind of wrench should be used for this step?"

The system will monitor the apprentice's work to ensure that the operation is proceeding normally. When the system becomes aware of an unexpected event, it will alter instructions to the apprentice to deal effectively with the new situation.

The system will be organized in such a way that the expertise to be held in the CBC can be encoded modularly, and updated easily. Information about new equipment should be easy to add. The system, and not the encoder of the system's expertise, should have major responsibility for the integration of independent items of knowledge.

Each of these capabilities is present in at least rudimentary form in the current NOAH system. With an appropriate representation for the system's knowledge, their achievement was, in fact, rather straightforward.

II A DESCRIPTION OF THE PROCEDURAL NET REPRESENTATION

A. Historical Perspective

The representation of knowledge within a computer memory is an issue which every attempt at artificial intelligence must face. For a number of years there has been considerable controversy in the artificial intelligence community over how this should be done.

The controversy concerned whether knowledge should be represented in declarative form or in procedural form. Some held that knowledge should only be stored in one or the other form. Later, a strong case was made for hybrid representations with both procedural and declarative components. These hybrids, called "frames" [25,41], "beings" [19], or "property lists" [20] reflect the general agreement of workers in the field that aspects of both representations are needed in all programs. While this is certainly true, the vague label of "frame" does mask an important underlying distinction between the kinds of knowledge that are used in problem solving.

On the one hand, basic knowledge about the task domain must be given to the computer. We shall call this kind of knowledge the domain knowledge. An example of domain knowledge is a characterization of the legal operations within the domain that transform one state into another. Other examples are knowledge about the ways in which individual actions can interrelate, and the interrelationships among the relations that describe the domain.

On the other hand, a problem solver develops another kind of knowledge as it plans about the effects of alternative sequences of actions. We shall call this kind of knowledge the plan knowledge. An example of plan knowledge is the ranked collection of alternative sequences of individual actions that the problem solver has under consideration for achieving a given goal state. Other examples are a characterization of the purpose of each action within an overall plan, and a measure of the cost or difficulty of each action.

These kinds of knowledge are clearly related, since the planned effect of a given sequence of actions depends on the domain-specific character of the individual actions. But the kinds of knowledge are used in very different ways.

Most early computer programs that performed at all impressively read in unit after unit of data and performed a fixed series of calculations on each one. Many programs that attempted to reason intelligently followed this paradigm of applying a fixed algorithm to different sets of data. They accepted unit upon unit of domain knowledge in declarative form, and processed it with a general-purpose algorithm. Examples of such programs include theorem provers (for example, Green's QA3 [15]) and language understanding systems using semantic nets (for example, Quillian's TLC [27]). Much of the plan knowledge in these systems was embodied in the code and the control structures of a general purpose data-massaging program.

For a computer system to deal with any meaningful domain of knowledge, such a representation will be inadequate. The declarative domain knowledge allows all the facts known to the system to be processed in a uniform way. But it requires that much of the plan knowledge be embedded in a rather general data massager

which will either be simple and therefore lack goal-directedness, or else be complex and therefore hard to update and improve.

In reaction to this problem, members of the Artificial Intelligence lab at MIT developed a philosophy about representation which is generally called the "procedural embedding of knowledge." Hewitt's PLANNER language [18] was developed as a vehicle for this embedding. The full PLANNER language is extremely complex, and was never implemented. A simplified version, MICROPLANNER, was developed by Sussman and Winograd [37], and was used with considerable success in a planner for a simple blocks world [42]. But there was a big problem with such a "simple" procedural representation. It did not have a good way of dealing with alternatives. They could only be explored by running out some control path until the program ran into trouble, then backing up to a place where an alternative had been picked, choosing another alternative, and running the program out again.

This blind approach to reasoning about alternatives, called "automatic backtracking," is required in MICROPLANNER because the language provides no way to represent plan knowledge in a form accessible to the programmer. Sussman and McDermott [36] proposed to deal with this problem by decoupling the state of the data base from the state of the control path. Then the data base would be used for domain knowledge, and the control paths would be used to represent plan knowledge. The control paths can be explicitly referenced in this approach. The result of this attempt, CONNIVER [22], has typically been used [35, 9] by saving multiple control environments that are queried during attempts at analyzing alternatives. The control environments of CONNIVER thus provide a means of

representing plan knowledge, but the representation is quite hard to use. Sophisticated programs using the CONNIVER representation have typically been extremely slow, since it is costly to use the control environments for plan knowledge.

The procedural embedding approach was overdone. It is true that procedures allow efficient goal-directed processing, and are easy to update. But it is difficult for a procedurally based knowledge system to characterize what knowledge is available to it. Furthermore, it is difficult to leave an appropriate record of the invocations of the procedures which represent the system's knowledge. A knowledge-based system that is engaged in a mixed-initiative interaction must know what it knows and know what it is doing if it is to have us believe that it "understands."

To encode NOAH's knowledge of the actions in its world, we have taken yet a third approach to the representation issue. We call our representation a procedural net. It is a strongly connected network of nodes, each of which may contain both procedural and declarative information. The procedural information is used to represent the domain knowledge, as was done by the "procedural embedding" school. But the plan knowledge is represented declaratively in the contents of the nodes and in the structure of the net itself. This enables NOAH to handle easily rather sophisticated queries about the state of its own reasoning processes. NOAH is, in a limited sense, self-aware.

The structure of the procedural net is the major source of NOAH's power in solving problems and monitoring their execution. Let us examine it in detail.

B. The Structure of a Procedural Net

The procedural net is a network of actions at varying levels of detail, structured into a hierarchy of partially ordered time sequences. By an action, we mean any operation that changes the state of the world. An action at a particular level of detail is represented by a single node in the procedural net. Each node contains procedural information, declarative information, and pointers to other nodes. The nodes are linked to form hierarchical descriptions of operations, and to form plans of action.

The information associated with each node is stored as a set of property-value pairs. A full list of the properties that a node can be found in Appendix A.

Each node in the procedural net may refer to a set of child nodes that represent more detailed subactions. When executed in order, the child nodes will achieve the effects of their parent. The parent-child relationships allow the system to represent and reason about actions in a hierarchical fashion.

Nodes at each level of the detail hierarchy are linked in a partially ordered time sequence by predecessor and successor links. Each such sequence represents a plan at a particular level of detail.

There are many different types of nodes. Each type represents an action (or a dummy action) with different characteristics. The types that represent actions will be discussed in the following section. Some of the more esoteric types will be discussed as they appear in the text.

Each node points to a body of code. The action that the node represents can be expanded to greater detail by evaluating the body. The evaluation will cause new nodes, representing more detailed actions, to be added to the net. It will also associate with these nodes the effects of the more detailed actions.

Associated with each node is an add list and a delete list. They contain symbolic expressions representing the changes to the world model caused by the action that the node represents. The add and delete lists are used to develop models of the world at each point in a plan. How this is done is discussed in detail in Section D below.

Together, the add and delete lists represent how the action that a node represents affects the world model. We shall sometimes refer to the add and delete lists together as the effects of a node. The effects are computed when the node is created, and may be updated while the plan containing the node is completed. No information from more detailed subactions is ever reflected in the effects of a node.

Each node of the procedural net thus contains two very different representations of an action. The add and delete lists provide a declarative representation of actions that is quite similar to that of STRIPS [13]. The body of code provides a procedural representation similar to that of PLANNER-like languages [2]. The declarative representation is used to model the action at the node's own level of detail. The procedural representation is used for generating more detailed subactions at levels of greater detail.

Figure 1 shows the graphic notation used here to display a node of a procedural net.

Node types are designated as follows:
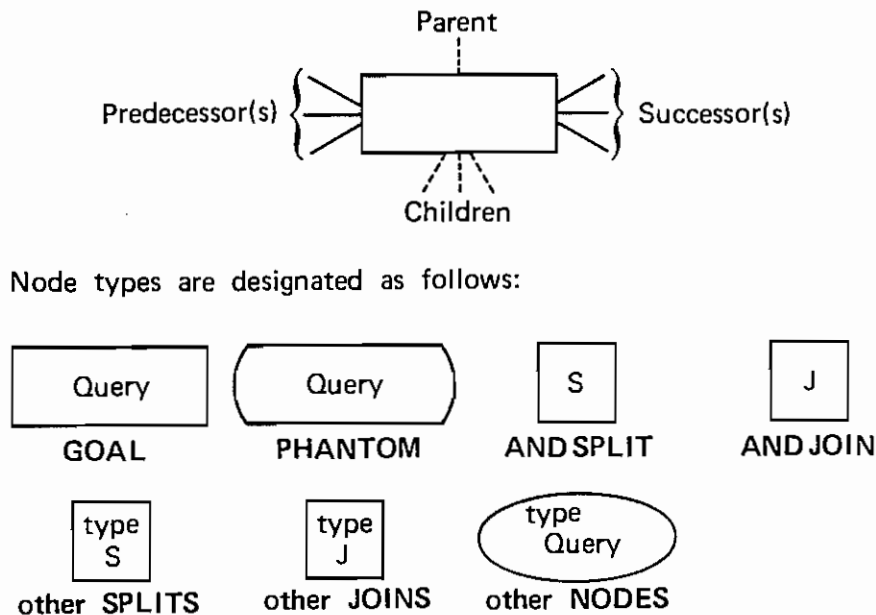
FIGURE 1   GRAPHIC REPRESENTATION OF A NODE

As a simple example, let us examine a procedural net representing a hierarchy of plans to paint a ceiling and paint a stepladder. The plan can be represented, in an abstract way, as a single node as shown in Figure 2a. In more detail, the plan is a conjunction, and might be represented as in Figure 2b. The major subgoals to paint the ladder and the ceiling are in parallel branches of the partial ordering. This means that they are both to be achieved after the split, and before the join, but that at this level of detail the representation does not specify whether one action should be performed before the other.

LEVEL 1

Paint the ceiling and paint the ladder

(a)

LEVEL 2

S

Paint the ceiling

Paint the ladder

J

(b)

LEVEL 3
(Before Criticism)

S

Get paint — Get ladder — Apply paint to ceiling +

Get paint — Apply paint to ladder −

J

(c)

LEVEL 3
(After Criticism
by Resolve Conflicts)

S

Get paint — Get ladder — Apply paint to ceiling

Get paint — J — Apply paint to ladder

(d)
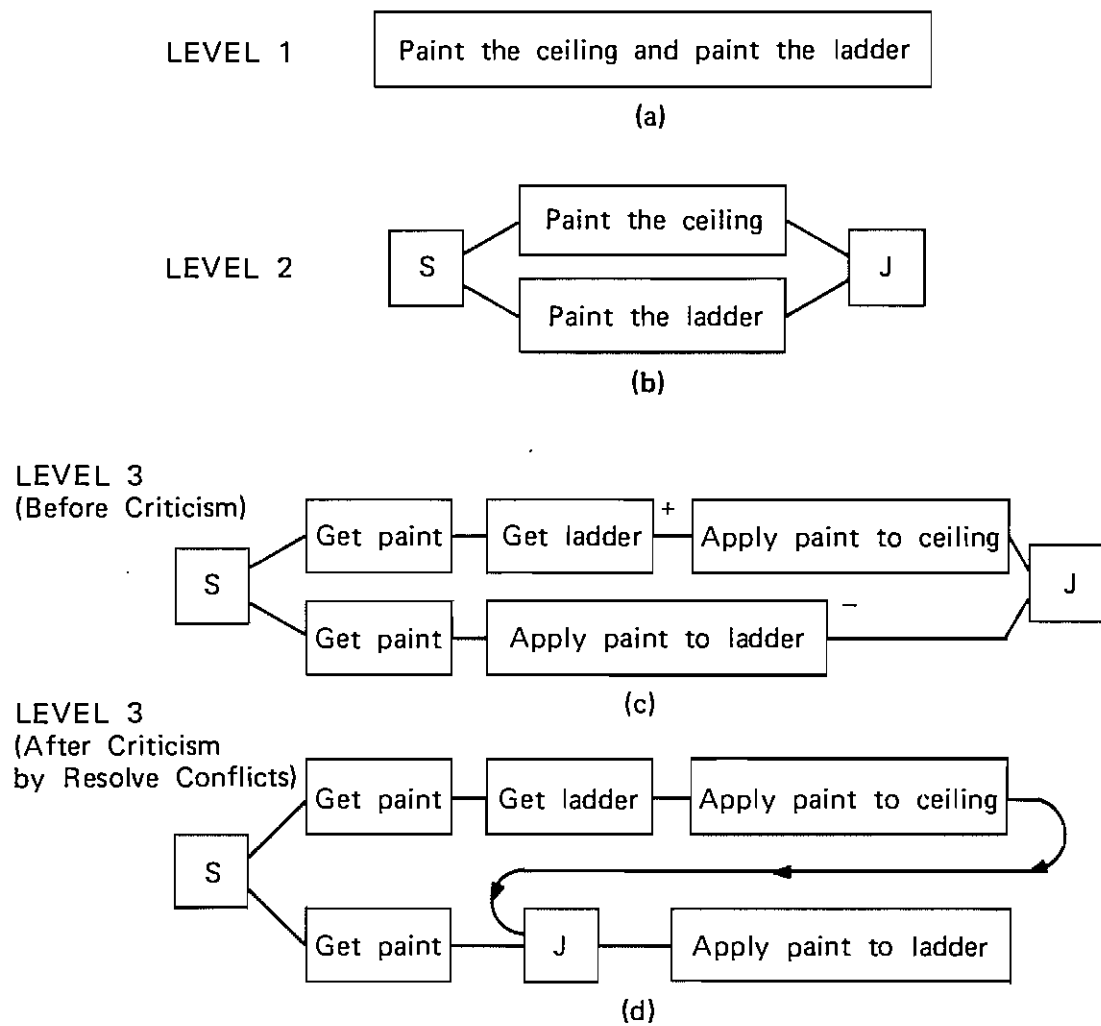
FIGURE 2   PROCEDURAL NET FOR PAINTING

The more detailed subplans to achieve these two goals might be "Get paint, get ladder, then apply paint to ceiling," and "Get paint, then apply paint to ladder," as depicted in Figure 2c. Note that the plan in Figure 2c is not a reliable one: it allows for the ladder to be painted before it must be used for painting the ceiling. A safer plan is depicted in Figure 2d. How NOAH transforms the unreliable plan into a safe one will be described in the next chapter.

For the sake of clarity, we will usually omit (as we have in Figure 2) the special PLANHEAD node that indicates the start of each plan. We will also often omit much of the information associated with each node from the pictorial representation. The add and delete lists, for instance, are not indicated in Figure 2. They are not hard to infer, however. For example, "Get ladder" will cause "Has ladder" to be added to the world model, and "Apply paint to ceiling" might delete "Has paint" from the world model.

The system infers certain important relationships from the parent-child links that indicate which nodes represent detailed expansions of other nodes. (These pointers are also omitted in Figure 2.) The system assumes that the purpose of every action but the last in such an expansion is to establish the truth of some expression in order to make the final action applicable. We will call both the expressions that are to be made true and the nodes that make them true preconditions. We will call both the last node in a more detailed expansion of a node and the pattern associated with the last node the purpose of the preconditions.

C. Encoding Domain-Specific Knowledge

Knowledge about the task domain is given to the system in procedural form, written in the SOUP (Semantics Of User's Problem) language. SOUP is an extension of QLISP [29] that is interpreted in a unique fashion*. The process of planning transforms this procedural knowledge into the hybrid procedural net form, which contains both procedural and declarative information, and which represents a hierarchy of solutions to the particular problem at hand.

We will first present the statements of SOUP that have been added to QLISP (we will call them P-statements). Then we will describe how the SOUP code is interpreted, and show some specific examples of SOUP code.

P-statements that refer to actions are:

PGOAL -- A PGOAL statement is of the form: (PGOAL query pattern APPLY team). Its meaning is similar to the QLISP GOAL statement. It has an additional argument, the query, that specifies a verbal request for the goal to be achieved. Evaluation of a PGOAL results in the insertion of a new node in the procedural net. If a true instance of the goal pattern is found in the world model, a PHANTOM node is created. If no true instance is found, a GOAL node is created. The add list of the node will contain an expression that is the result of instantiating the pattern when the PGOAL statement is evaluated. The body of the node will be the list of procedures specified by the team.

---

*We will presume that the reader is familiar with QLISP or some other PLANNER-like language. Bobrow and Raphael [2] provide a survey of some of these languages. In QLISP and SOUP, variables are indicated by the prefix ← or $. The ← prefix indicates that the variable is to be assigned a new value; the $ prefix indicates the previous value of the variable.

PBUILD -- A PBUILD statement is of the form: (PBUILD class-name query (ITERATE...)). It specifies an action that will build up a class of objects. The query is a verbal request to build up the class. The ITERATE statement contains an arbitrary body of code that specifies the subactions involved in processing one element of the class. Evaluation of a PBUILD statement results in the creation of a BUILD node in the procedural net.

PBREAK -- PBREAK has the same syntax as PBUILD. It specifies an action that iterates through a preexisting class of objects. Evaluation of a PBREAK statement results in the creation of a BREAK node in the net.

PAND -- A PAND statement is of the form: (PAND exp-1 exp-2 ... exp-n). PAND specifies a collection of expressions which may be evaluated independently. Execution of a PAND statement results in a n-way branching in the partially ordered time sequence of the plan at the current level of detail in the procedural net. The branching is delimited by an ANDSPLIT and an ANDJOIN node.

POR -- POR has the same syntax as PAND. It specifies a collection of expressions, only one of which need be evaluated without failure. Execution of a POR statement results in a n-way branching in the time sequence of the plan at the current level of detail. Only one branch need be included in a final plan. The branching is delimited by an ORSPLIT and ORJOIN node.

P-statements that refer specifically to the world model are:

PIS -- A PIS statement is of the form: (PIS exp). It searches for an instance of the expression exp that is true in the current world model. (How this search is

performed is described in Section D below.) If none is found, it causes a failure condition.

PINSTANCES -- A PINSTANCES statement is of the form: (PINSTANCES exp). It operates like PIS except that it returns all the instances of exp that are true in the current world model.

PASSERT -- A PASSERT statement is of the form: (PASSERT exp). It makes exp be true in the current world model and places exp on the add list of the most recently generated node in the procedural net. PASSERTs are used to specify the equivalent of the secondary adds of the STRIPS problem solver [12].

PDENY -- PDENY is similar to PASSERT. It makes exp be false and adds exp to the delete list of the current node.

PRECLUDE -- A PRECLUDE statement has the form: (PRECLUDE variable value). It is used to preclude the binding of a variable to a specific value when the next PGOAL statement queries the data base. This forces the PGOAL statement to seek an alternative expression in the data base, or to set up a genuine GOAL node to achieve its truth.

When a P-statement that refers to an action (i.e., PGOAL, PBUILD, PBREAK, or PAND) is evaluated, it does not cause an arbitrarily deep computation, as would most PLANNER-like languages [2]. Rather, the action is simulated at an abstract level and the world model is updated as if the deep computation had been done and the action were accomplished in full detail. The information necessary to continue the computation to further depths is stored as the body of code associated with the

new nodes that are created in the procedural net. For example, when a PGOAL statement is evaluated (thus simulating the action that achieves the goal), the team of functions associated with the statement is placed as the body of the new node representing that goal. When a PBUILD or PBREAK statement is evaluated, the ITERATE clause is stored as the new node's body. This type of evaluation results in the creation of hierarchies of plans of increasing detail. This scheme thus extends the ability to do hierarchical planning as was done by ABSTRIPS [31] from a syntactically oriented declarative representation to SOUP's semantically oriented procedural representation.

As an example, let us examine some SOUP functions for a set of simple blocks problems. The complete semantics of the actions of this domain are listed in Appendix B. The simplest problems can be solved with just the functions shown in Figure 3. Let us look at .CLEAR, for example. The code says, "If the variable X is TABLE, then it is already 'clear'. Otherwise, see if some block Y is on X. If so, clear Y and then remove Y by putting it somewhere else."*

Now, in any PLANNER-like language, the subgoal to clear Y would be immediately attacked. If there were a large tower of blocks on top of Y, they would all be cleared at this time. When NOAH evaluates the SOUP code, however, it simply asserts (CLEARTOP $Y) and then goes on with the evaluation of CLEAR. It also places a node in the procedural net whose type is GOAL, whose pattern is (CLEARTOP $Y), and whose body is a list containing the single function CLEAR. This is sufficient information to enable NOAH to fire up the subgoal to clear Y when it needs to plan to greater detail.

---

*Note that this is a very simple-minded interpretation of "clear." A more sophisticated procedure might find space on X for an additional block.

FIGURE 3

SOUP CODE FOR SIMPLE BLOCKS PROBLEMS

```
(CLEAR
   [QLAMBOA
     (CLEARTOP ←X)
     (OR
       (EQ $X (QUOTE TABLE))
       (QPROG
          (←Y)
          (ATTEMPT (PIS (ON ←Y $X))
               THEN (PGOAL (Clear $Y)
                           (CLEARTOP $Y)
                           APPLY
                           (CLEAR))
                    (PRECLUDE ←Z $X)
                    (PGOAL (Put $Y on top of ←Z)
                           (ON $Y ←Z)
                           APPLY NIL))
                    (PDENY (ON $Y $X))
          (RETURN])

(PUTON
   (QLAMBOA
     (ON ←X ←Y)
     (PANO
       (PGOAL (Clear $X)
               (CLEARTOP $X)
               APPLY
               (CLEAR))
       (PGOAL (Clear $Y)
               (CLEARTOP $Y)
               APPLY
               (CLEAR)))
     (PGOAL (Put $X on top of $Y)
             (ON $X $Y)
             APPLY NIL)
     (PDENY (CLEARTOP $Y))))
```

If there were no blocks on Y, and (CLEARTOP $Y) were known to be true, the GOAL statement of a PLANNER-like language would be satisfied and the program would go on. Similarly, when NOAH evaluates a PGOAL statement which is satisfied in the current world model, it places a node in the procedural net of type PHANTOM. A PHANTOM node is just like a GOAL node except that it will not be expanded to greater detail.

D. A Distributed World Model and an Approach to the Frame Problem

The procedural net is a highly structured representation for actions. Various aspects of the problem solving and execution monitoring functions of NOAH analyze the effects of aggregations of actions in different ways. It is therefore important to have an easy method for constructing and updating world models for arbitrary aggregations of actions. The procedural net allows for this by associating a partial world model with each action. The partial model reflects just the changes to the world caused by the individual action. The partial world models are layered upon one another, in an order determined by the time sequence of the actions, to form a complete world model. The links that order the layers of the model are precisely the same links that order the actions in time sequence.

For example, Figure 4 shows the add and delete lists that might be associated with each node in the most detailed plan in the painting example discussed earlier. A query of the world model (in particular, the evaluation of the statement: (PINSTANCES ←X)) from a point just before the "Apply paint to ladder" node in Figure 4a would find just the assertion (ONHAND LADDER-PAINT). The same query made from the same point in Figure 4b, which is identical to Figure 4a except for an
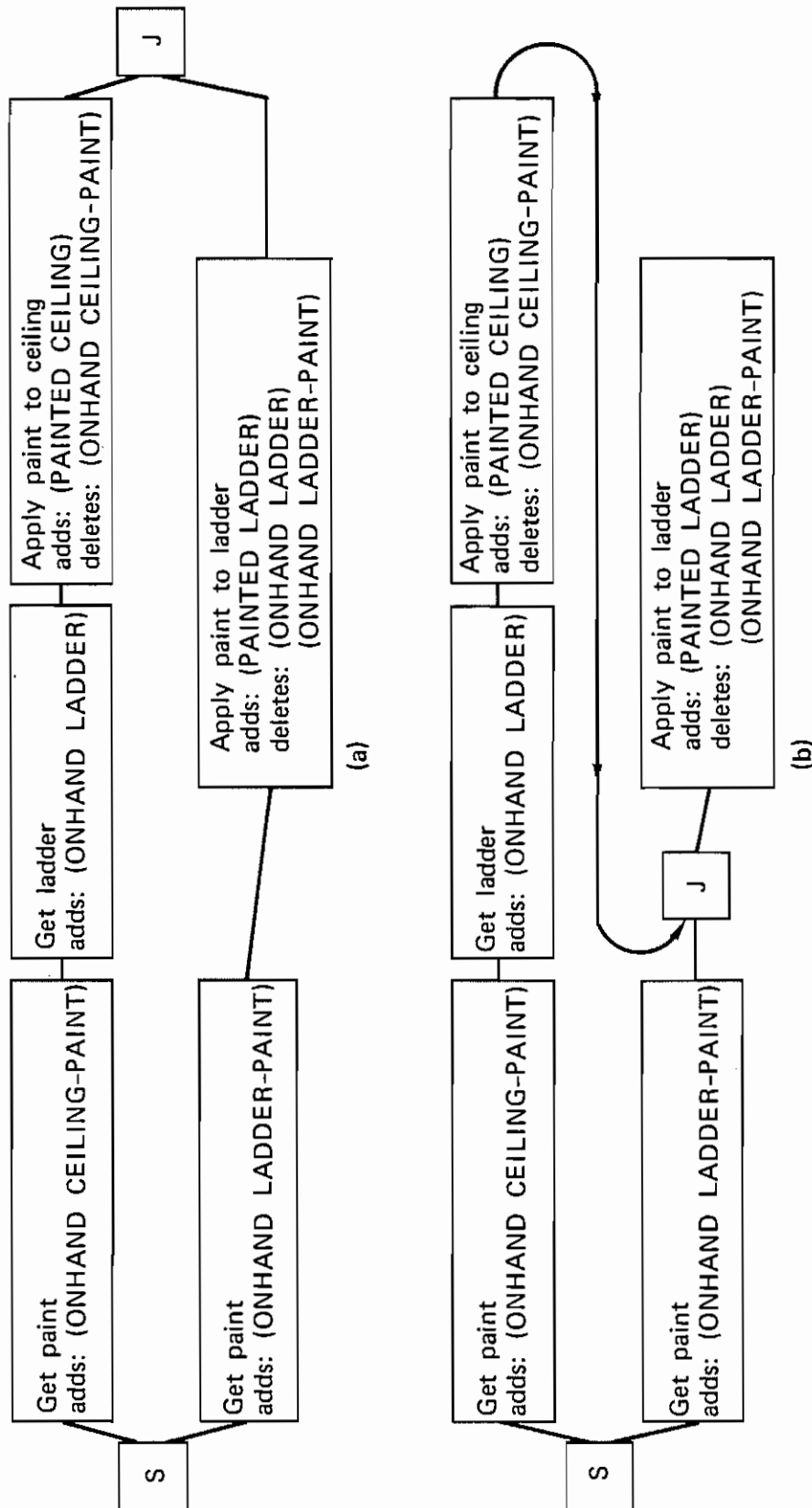
FIGURE 4   ADD AND DELETE LISTS FOR PAINTING PLANS

alteration in the partial ordering, would find the assertions (ONHAND LADDER-PAINT), (PAINTED CEILING), and (ONHAND LADDER). For lengthy plans and large world models, this method of maintaining and querying a world model can be quite inefficient. Typically, most of the facts in an initial model will not be altered during the course of a given plan. (This state of affairs is a broader view of what McCarthy and Hayes [21] termed the "frame problem": that most facts in a world model are not altered by applying a single action.) If the distributed world model were used in its pure form, most queries would have to be carried all the way back to the PLANHEAD node that begins each plan before an answer could be determined. This is essentially what happens in the modelling schemes used by Waldinger [39] and Warren [40].

NOAH uses a hybrid approach, so that all the expressions whose truth value has never changed (i.e., the frame axioms) are accessed through a global world model. When an expression's truth value is changed, it is removed from the global model. Its new truth value is indicated by placing it on the add or delete list of the node that is altering it. Its former truth value is indicated by inserting the expression on the add or delete list of the PLANHEAD node of the plan being generated. The model query algorithm always checks the global world model first. If a truth value can be determined from this check, the querying process is finished. If not, the distributed world model contained in the current plan is checked.

This method of maintaining and accessing a world model thus provides an efficient means of dealing with frame axioms while also providing a very flexible method of modelling the changes caused by actions and plans of actions.

E. A Very Simple Example

To obtain a better understanding of how the representation is used, let us examine in detail how NOAH solves a very trivial problem. (The casual reader may safely skip this section.) The system knows only of two blocks. In the initial state, Block B is on Block A. The goal is to achieve a state in which Block A is on Block B, as depicted in Figure 5. The problem can be solved using just the node expansion process which has been described in section C. The problem solving techniques used by NOAH for more sophisticated problems will be discussed in the next chapter.

Initial State:
(ON B A)
(CLEARTOP B)

Goal State:
(ON A B)
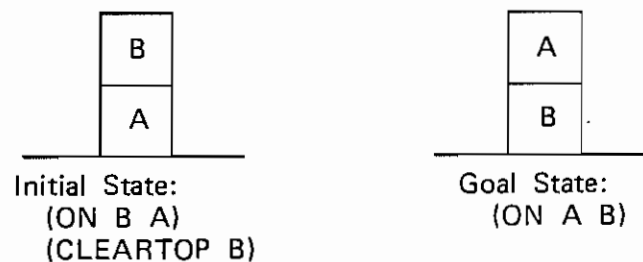
FIGURE 5   A VERY SIMPLE PROBLEM

An initial world model is defined by ASSERTing and DENYing expressions as in ordinary QLISP. For the example, the following QLISP statements cause the initial world model to be set up:

    (ASSERT (ON B A))
    (ASSERT (CLEARTOP B)).

The system is invoked with the goal: (ON A B).

It builds an initial PLANHEAD node in the procedural net. This is followed by a

GOAL node. Its pattern is (ON A B). Its query is (Achieve (ON A B)). Its body is the value of the system variable $ALLFNS, which the definer of the domain has set to a list of all the top-level functions in the domain. In this case, this is the list of programs: (PUTON CLEAR). (These programs were listed in Figure 3.) The node's context is a descendant of the global context in which the original world model was specified. Its add list consists of the single expression: (ON A B). This single node represents a high-level plan to achieve (ON A B). The plan is depicted in Figure 6.
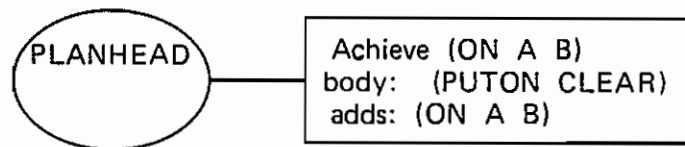
FIGURE 6   EXAMPLE PROCEDURAL NET BEFORE EXPANSION

The simple plan is now expanded. This is done by expanding each node in the partially ordered time sequence of the plan. Ties are broken arbitrarily. First, the PLANHEAD node is expanded. Since it has no body, a child is created that is a simple copy. The GOAL node is next expanded. Each of the functions in its body is applied in turn to the pattern (ON A B), using the standard QLISP pattern matching algorithm. PUTON, the first function in the body, is applicable. PUTON specifies that three statements are to be evaluated in sequence: a PAND, a PGOAL, and a PDENY. The PAND is evaluated first. It causes a SPLIT node to be added to the new plan. Then, each of the PGOALs within the PAND is evaluated. Each causes a new node to be added. The new nodes have (CLEARTOP A) and (CLEARTOP B) as their patterns. Both have the function CLEAR as their body. The node referring to (CLEARTOP A) is a GOAL node, since (CLEARTOP A) is not true in the current world

model (the model associated with the new SPLIT node) when the PGOAL statement is evaluated. The node referring to (CLEARTOP B) is a PHANTOM node, since (CLEARTOP B) was asserted in the initial world model, and has not been denied since. Finally, the evaluation of PAND is completed by inserting a JOIN node after the two new nodes.

Next, the PGOAL statement of PUTON is evaluated. It causes a new node whose pattern is (ON A B) to be added to the new plan after the JOIN node. The node is a GOAL node since (ON A B) is not true in the current world model. The node has no body; the action it represents is a primitive in the domain. It might represent an actual high level command to a robot arm.

Finally, the PDENY statement is evaluated. It causes (CLEARTOP B) to be deleted from the global context, and inserted on the add list of the PLANHEAD nodes at the current level. It also causes (CLEARTOP B) to be inserted on the delete list of the recently created GOAL node.

Now, the expansion of the higher level GOAL node has been completed. When the expansion of a node is completed, the add and delete lists of the newly created last child are augmented by the expressions on the add and delete lists of the parent. The last child node in the expansion of a parent node inherits its parent's effects, since by the time the last child has been executed, the entire higher-level action will have been performed. In this case, the GOAL node that is the last child already has (ON A B) on its add list, so nothing extra is added.

Figure 7 shows the procedural net with the new, more detailed plan.
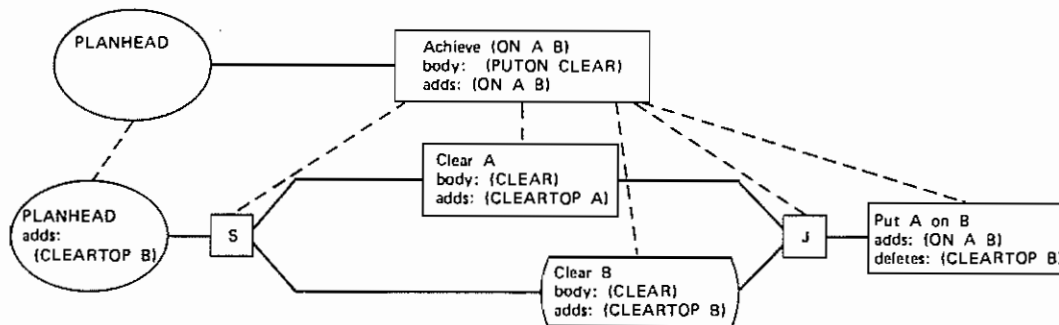
FIGURE 7   EXAMPLE PROCEDURAL NET AFTER ONE STAGE OF EXPANSION

This new plan is now expanded. First, a new PLANHEAD node is created. Then the SPLIT node is simply copied since it has no body. The system begins work on the upper branch of the conjunction. The GOAL node whose pattern is (CLEARTOP A) is now expanded. Its body is the single function CLEAR, and it is applicable to the pattern. CLEAR checks to see if A is the TABLE, finds it is not, and then queries the model to see what is on A. The context of the node contains the assertion (ON B A), so the local variable $Y is bound to B. Then a PGOAL, a PRECLUDE, another PGOAL, and a PDENY are evaluated in sequence. The first PGOAL results in the creation of a PHANTOM node with (CLEARTOP B) as its pattern and the function CLEAR as its body. The PRECLUDE statement forbids the following PGOAL to succeed by finding that (ON B A) is true in the current world model. Thus, the final PGOAL results in the creation of another GOAL node. The unbound variable ←Z is treated specially and is bound to an unspecified object Object1. (We will explain the treatment of unbound variables in the next chapter.) The new GOAL node's pattern is thus (ON B Object1). Its body is empty since the PGOAL statement specified no APPLY team, and thus this action represents a primitive in the system.

Finally, the PDENY causes (ON B A) to be inserted into the delete list of the new GOAL node.

This completes the expansion of the GOAL node whose pattern was (CLEARTOP A). So the add and delete lists of the last child are augmented by the adds and deletes of the GOAL node. In this case, the expression (CLEARTOP A) is added to the add list, since putting B on Object1 does clear A.

The PHANTOM node whose pattern is (CLEARTOP B) is simply copied to the lower level.

The JOIN node in the plan in Figure 7 is now expanded. It has no body, and so is simply copied to the lower level.

Finally, to complete the expansion of the plan, the GOAL node whose pattern is (ON A B) is expanded. Again, it has no body (it is a system primitive) and so is simply copied. This completes the expansion of the plan in Figure 7. The net including the new plan is shown in Figure 8.
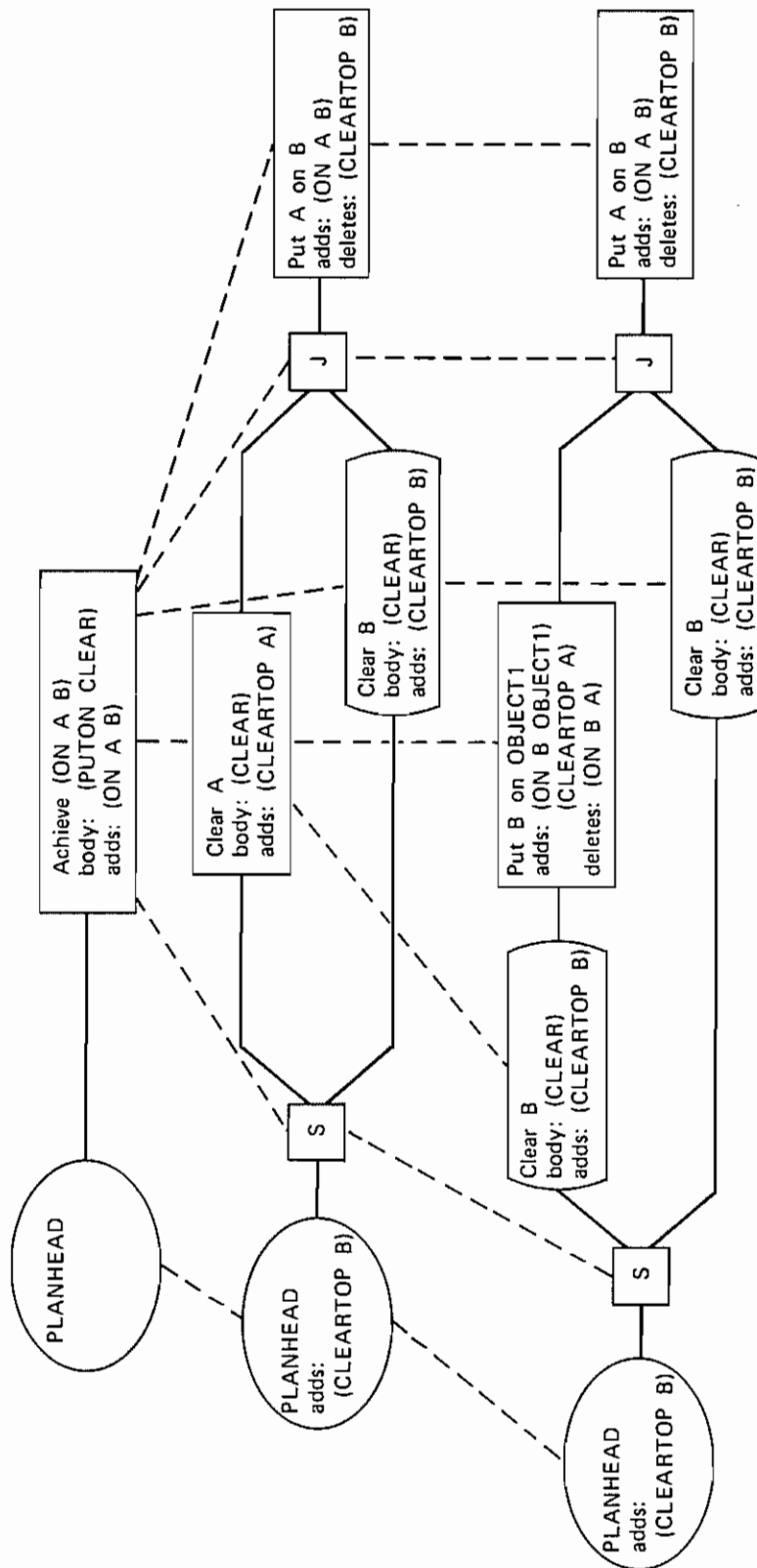
FIGURE 8   EXAMPLE PROCEDURAL NET AFTER TWO STAGES OF EXPANSION

F. Comparison with Other Current Work

A number of suggestions have been made in the past few years for structuring procedural and declarative knowledge together in useful ways. Of major influence among these has been the concept of a "frame" as a unit of organization of knowledge, as proposed by Minsky [25], Winograd [41], and others. NOAH can be usefully viewed as a problem solver that utilizes action frames of a restricted variety for specific purposes. Each node in the procedural net is an aggregation of information about an action. Some of its "slots" refer to particular values or expressions; others are pointers to other frames. The body of code is a procedural specification of detailed knowledge about the action. This procedural specification is transformed into a partially instantiated action frame by the node expansion process. Most of the problem solving work is the building, analysis, and interrelating of these action frames. Most of the execution monitoring work is the carrying out of behavior as characterized by the frame structure.

The procedural net provides a number of efficiencies over a more general framelike representation. First, the semantics of actions are specified in a strictly procedural form. This is a natural and concise way for the definer of a domain to describe them. The problem solving system develops a more complex frame representation itself as it goes along. Furthermore, the ways in which the action frames are allowed to interact are very restricted. This allows a set of special modules, called critics, to do a reasonably complete job of dealing with interactions among the action frames. Critics will be described in much more detail in the following chapter.

The information in the action frames of the procedural net can come from only two sources: the procedural specification of the semantics of an action, and the current environment in which the procedures are simulated. This reduces the power of the frame-like approach, and many kinds of information that could be of great use within the rich structure of the procedural net representation cannot currently be inserted into the frames. Section A of Chapter VI discusses some of these.

Although the procedural net is a unique approach to representing actions, it is not the only one that combines both procedural and declarative aspects. In addition to frame-like systems as described above, Scragg [32] and Rieger [30] have implemented representations for actions that combine both approaches. Their representations are essentially semantic networks representing programs. The networks can be interpreted by a special executive, and thus treated procedurally, or they can be analyzed as a data structure, and thus treated declaratively. This approach allows much more information to be extracted from the declarative representation, but both the procedural and the declarative uses of the information are much slower.

Let us now explore how NOAH uses the information stored in the procedural net to solve problems and monitor their execution.

III GENERATING PLANS OF ACTION

A. Historical Perspective

Almost every intelligent operation requires some degree of problem solving ability. Most artificial intelligence programs have task-specific problem solvers built into them. In addition, researchers have investigated the process of solving problems per se. Quite a few approaches have been attempted over the years, with very different characteristics. This section will attempt to discuss the differing approaches by casting them into a common paradigm.

Problem solving may be thought of as a process of searching through a space of possibilities for a point (a solution state) or path (a plan) that satisfies an externally imposed criterion (the problem). Let us define the solution space to be that subset of the space of possibilities within which the planner believes the plan to lie. (There is, of course, no guarantee that the solution space does in fact contain the solution.) Within this framework, we may define the process of problem solving as a development of constraints that progressively narrow the solution space. When a problem solver is first given a problem, the solution space is as large as the space of possibilities. The problem solving task is to progressively narrow the solution space. This is done by applying constraints to the best current solution space.

This formulation of the task of problem solving is far broader than the more traditional one of performing trial-and-error search through the solution space to

find a path to a given goal. The use of search strategies is seen here as simply one approach to constraining the search space.

Many of the successful general problem solving systems (for example, GPS [8] and STRIPS [13]) used a search strategy called means-ends analysis. It works by examining the goal state and determining a subset of operators that might lead to the goal state. The preconditions of these relevant operators constrain the solution path to traverse particular subgoal states, which in turn can be examined. The system works backward until a path is defined from the initial state to the goal state.

The trouble with a pure means-ends analysis strategy is that the planner often has no good source of information about which relevant operators are most likely to lead to a solution. So the strategy is subject to the effects of a combinatorial explosion of choice points as the length of the plans becomes nontrivial.

The thrust of much of the work on general problem solving systems in the last few years has been on developing additional ways of constraining the search done by a simple means-ends analysis algorithm. These approaches include debugging almost-right plans, hierarchical planning, extensive constraint analysis, and progressive plan modification.

The basic idea underlying all of these approaches is that it is too expensive for a planner to analyze all the constraints on a problem solution at one time. What these approaches do, in one way or another, is to use only a small subset of the possible constraints to reduce the solution space as quickly as possible. Then a

final solution can be found by an expensive evaluation of all the constraints. Since the solution space is small, the expense of the detailed search is easier to bear.

There is a potential danger in this approach, though. The pure means-ends analysis strategy has a well-defined bound on its worst-case behavior. The chaining together of actions into partial plans is done in a basically breadth-first manner, and so an n-step plan will be found no later than after all plans of n or fewer steps. The new approach has no bound on its worst-case behavior. If the cheap analysis constrains the solution space to a portion of the search space that doesn't contain a solution, the planner may do arbitrary amounts of work before it discovers the mistake. Thus, in addition to using as little information as possible to constrain the solution space, a sophisticated planner must avoid constraining the solution space prematurely on the basis of insufficient information. Furthermore, it must have a means for dealing with the possibility that the high level constraints have eliminated the solution to the problem from the solution space.

Let us now take a brief look at some of the major new efforts in problem solving in the light of this analysis. A more detailed comparison will be presented after NOAH's planner has been explained in detail.

The HACKER system for planning in the blocks world [35] cheaply narrows the solution space by making a "linear assumption." That is, the system assumes that to solve a conjunction of goals, each one may be solved in turn. This won't provide a solution in most cases, of course, but it does provide a lot of information about how the pieces of the correct solution would act together. By making relatively minor modifications to a cheaply produced first guess, the system is able to develop

solutions to problems rather directly. HACKER often overconstrains the solution at first, but it has a reasonable model of what the higher level constraints do and so can recover from the problem.

The ABSTRIPS planning system [31] progressively narrows the solution space by solving the problem using simpler, less constrained operators. When a solution is found, the system will have defined islands in the space of possibilities that must be traversed if the problem can be solved by a detailed version of the higher level plan. The system now solves the smaller problems of navigating between the islands, using operator descriptions that are more constrained. This process continues until the fully constrained operators are used. By solving a hierarchy of simpler problems, ABSTRIPS is relatively insensitive to the combinatorial explosion. But its model of the relationships among the various levels of plan is contained in its control structure in an unusable way. Thus ABSTRIPS is prone to flounder if the higher level solutions were wrong.

NOAH performs its planning in a hierarchical fashion like ABSTRIPS, but the hierarchy is determined by the calling structure of the SOUP code rather than by a predetermined definition of operator hierarchies. In addition to trying to reduce the solution space with as little work as possible, NOAH tries very hard not to overconstrain the solution space. It uses the structure of the procedural net to retain information about available options in the developing plan. Plans are stored in the net as a partially ordered time sequence, rather than as a total ordering that traditional problem solvers use. In this way the system avoids constraining the order of actions unless there is good reason to do so. Furthermore, the hierarchical

nature of the procedural net itself allows the system to retain a good deal of information about the structure of the developing plan. Thus, if the system should find that it has overconstrained, sufficient information has been saved in a readily usable form so that intelligent recovery can take place.

B. The Basic Planning Algorithm

Initially, NOAH is given a goal to achieve. NOAH first builds a procedural net that consists of a PLANHEAD node and a single GOAL node to achieve the given goal. This node's body is constructed from the value of the system parameter $ALLFNS and from a set of general-purpose functions for dealing with expressions beginning with AND, OR, and NOT. The value of $ALLFNS is set by the definer of the domain to a list of all the top-level SOUP functions in the domain. This single node represents a plan to achieve the goal at a very high level of abstraction. This one-step plan may then be expanded.

The planning algorithm of the NOAH system is simple. Its input is a procedural net. It expands the most detailed plan in the net by expanding each node of the plan in turn. In addition to building a detailed model of the effects of each action in the plan, the expansion of each node will produce child nodes. Thus by expanding the plan, a new, more detailed plan will be created.

The individual subplan for each node will be correct, but there is as yet no guarantee that the new plan, taken as a whole, will be correct. There may be interactions between the new, detailed steps that render the overall plan invalid. For example, the individual expansions involved in generating the plan in Figure 2c

from that in Figure 2b are correct, yet the overall plan is invalid, since it allows for painting the ladder before painting the ceiling.

Before the new detailed plan is presumed to work, the planning system must take an overall look at it to ensure that the local expansions make global sense together. This global examination is provided by a set of critics. The critics serve a purpose somewhat similar to that of the critics of Sussman's HACKER [35], except that for NOAH they are constructive critics, designed to add constraints to as yet unconstrained plans, whereas for HACKER they were destructive critics whose purpose was to reject incorrect assumptions reflected in the plans.

The algorithm for the planning process, then, is as follows:

(1) Expand the most detailed plan in the procedural net. This will have the effect of producing a new, more detailed plan.

(2) Criticize the new plan, performing any necessary reordering or elimination of redundant operations.

(3) Go to Step 1.

Clearly, this algorithm is an oversimplification, but for now we may imagine that the planning process continues until no new details are uncovered. As other features of NOAH are described, modifications to this algorithm will be presented*.

---

*One simplification that is retained in the current version of NOAH is that all the children of a node will be placed in the partial ordering ahead of all the children of a subsequent node. This simplifies the operation of the critics, but may cause NOAH to fail to find a valid plan where one exists.

C. Constructive Critics

The concept of constructive criticism is central to NOAH's approach to problem solving. This section will provide a justification for the reliance on the criticism approach. Then, examples of general-purpose and domain-specific critics will be presented.

1. The Rationale Behind Criticism

The constructive critics of NOAH constitute that portion of the system that concerns itself with the interactions among individual actions. The architecture of the NOAH system is unusual in that this function is explicitly separated from other aspects of the problem solving process. What advantages does this provide?

The major reason for the use of critics lies in the nature of hierarchical planning. The basic strategy of hierarchical planning is to create a cheap, if incorrect, plan by throwing much information away. Then the cheap plan is expanded into a more detailed plan. The process of expansion continues, building ever more detailed plans until a sufficiently accurate one has been built. The process of expanding a plan to a greater level of detail can be made very simple. For NOAH, the expansions to a greater level of detail are all specified only in terms of local considerations. Since the considerations are local, the search involved in any particular expansion will be relatively small. But since all the expansions are performed strictly on local considerations, a global overview is necessary to ensure that the individual expansions make sense together. It is much more efficient to do a series of local searches, followed by a global search for specific types of interactions, than to perform a global search at each step.

The criticism approach also allows for much greater modularity within the semantics of the task domain. If the model of each action had to include information about all the potential interactions with other actions, it would be impossible to encode domains of any complexity. The number of individual interactions will tend to grow much faster than the number of individual actions. Inserting a model of a new action would entail specifying many interactions along with the new action. Even worse, it would require revising the models of many actions that had already been modelled.

The use of critics allows individual actions to be specified without regard for their effect on every other action in the domain. A computer expert can thus be built from a collection of more easily constructed micro-experts rather than as a monolithic complex entity. Looked at another way, it allows information about a kind of interaction to be specified in a single module rather than being distributed throughout all the individual actions.

Another advantage of the criticism approach is that it segregates those constraints derived from local considerations from those that arise from more global restrictions. The local restrictions that are based simply on a particular action's semantics can be encoded directly into the SOUP code describing the action. These constraints will then be applied well before the more global ones that the critics provide. The search space can thus be reduced as much as possible by relatively cheap, local constraints without forcing possible wrong choices from premature application of global constraints.

2. General Purpose Critics

a. The "Resolve Conflicts" Critic -- The Resolve Conflicts critic examines those portions of a plan that represent conjuncts to be achieved in parallel. In particular, it looks at the add and delete lists of each node in each conjunctive subplan. If an action in one conjunct deletes an expression that is a precondition for a purpose* in another conjunct, then a conflict has occurred. The purpose is endangered because, during execution, its precondition might be negated by the action in the parallel branch of the plan. (An implicit assumption being made here is that all of a purpose's preconditions must remain true until the purpose is executed.) The conflict may be resolved by requiring the endangered purpose to be achieved before the action that would delete the precondition.

For example, the painting plan depicted in Figure 2c contains a conflict. "Apply paint to ladder" will effectively delete "Has ladder," which is on the add list of "Get ladder." In such a situation, a conflict would occur, since "Has ladder" is a precondition of "Apply paint to ceiling." The conflict is denoted in the pictorial representation by a plus sign (+) over the precondition and a minus sign (-) over the step that violated it. The conflict can be resolved by requiring that the endangered subgoal ("Apply paint to ceiling") be done before the violating step ("Apply paint to ladder"). If the conflict were resolved in this manner, the resulting plan would appear as in Figure 2d.

A more difficult conflict to deal with occurs if each of two conjunctive purposes has a precondition that denies the other purpose. This is a special case of

---

*"Purpose" is used in the technical sense defined in Chapter II, Section B.

the kind of conflict described above. This conflict is not linearizable, since no linear order of the purposes will achieve the overall goal. It is treated by a special critic, described in Subsection d below.

Another type of conflict occurs if an action deletes an expression that is a precondition for a subsequent purpose. In this case, the precondition must be re-achieved after the action that deletes it.

Conflicts of this type are very easy to spot. The critic simply builds a table of multiple effects (which we shall call a TOME). This table contains an entry for each expression that was asserted or denied by more than one node in the current plan. A conflict is recognized when an expression that is asserted at some node is denied at a node that is not the asserting node's purpose.

Note that a precondition may legally be denied by its own purpose. For example, to put Block A on Block B, B must have a clear top. This precondition will be denied by the action of putting A on B.

b. The "Use Existing Objects" Critic -- In addition to specifying the right actions in the right order, a complete plan must specify the objects that the actions are to manipulate. For NOAH, this specification is accomplished by binding the unbound variables (those prefixed by a left arrow) in the PGOAL statements of the SOUP code.

During the course of planning, NOAH will avoid binding a variable to a specific object unless a clear best choice for the binding is available. When no specific object is clearly best, the planner will generate a formal object to bind to

the variable. The formal object is essentially a place holder for an entity that is as yet unspecified. The formal objects described here are similar in spirit to those used by Sussman in his HACKER program [35], and to the uninstantiated parameters in relevant operators as used by ABSTRIPS [31].

The strategy of allowing actions with unbound arguments to be inserted into a plan has several advantages. First, it enables the system to avoid making arbitrary, and therefore possibly wrong, choices on the basis of insufficient information. Furthermore, it allows the system to deal with world models that are only partially specified by producing plans that are only partially specified.

However, after a plan has been completed at some level of detail, it may often be improved by replacing a formal object by some object that was mentioned elsewhere in the plan. The Use Existing Objects critic will replace formal objects by real ones whenever possible. This may involve merging nodes from different portions of the plan, resulting in reordering or partial linearization.

For example, a more detailed expansion of the painting plan might specify putting the ladder at Place001 to paint it, and at Under-Ceiling for painting the ceiling. The Use Existing Objects critic would optimize the plan by replacing Place001 with Under-Ceiling.

c. The "Eliminate Redundant Preconditions" Critic -- During the simulation phase of the planning process, every precondition that is encountered is explicitly stored in the procedural net. This is so that the critics will be able to analyze the complete precondition-purpose structure of each new subplan. But after the other

critics have done their work, and the plan has been altered to reflect the interactions of all the steps, the altered plan may well specify redundant preconditions.

For instance, in our painting example, "Get paint" appears twice in the plan. This critic recognizes the redundancy by examining the same TOME that was used by Resolve Conflicts. The extra preconditions are eliminated to conserve storage and avoid redundant planning at more detailed levels for achieving them.

d. The "Resolve Double Cross" Critic -- A special kind of conflict occurs when each of two conjunctive purposes denies a precondition for the other. This kind of conflict, which we call a "double cross," cannot in general be resolved by any linearization of the parallel subplans. The system must be creative and propose additional steps that will allow the two purposes to be achieved at the same time.

A number of approaches are possible for dealing with this case. At the worst, the system could simply replan to achieve the conjunction, with the constraint that the plan had to be linear. A better approach is to try to use as much as possible of the existing plans. The mutual conflict could be resolved by altering each subplan to be impervious to the effects of the other subplans. Another possibility is to make each subplan innocuous to the others. The latter approach has been followed here.

The strategy used in making each subplan innocuous is to look at the assertions or denials that led to the conflict. The system determines what variable bindings during the execution of the SOUP code caused the particular assertion or

denial to be made. Then, it inserts appropriate steps in the plan to ensure that the variables are bound differently at the time of the assertion or denial*. Thus, a different expression will be asserted or denied, so the particular conflict that occurred last time won't happen again. The remainder of the plan must be expanded again, taking the new steps into account, to be sure that no new conflict was introduced in the course of fixing the old one. If a new conflict is introduced, the current system gives up.

e. The "Optimize Disjuncts" Critic -- The mechanism for choosing a particular program to apply to the pattern of a GOAL node permits a kind of disjunctive processing. A choice among disjuncts is made at the time the node is expanded, and the current version of the system never deals with the other alternatives again. But there are times when a choice must be deferred until more information is developed at more detailed levels of the plan. For this case, a POR statement is provided in SOUP, to allow the system to deal with explicit disjuncts.

Parallel subplans are used to represent disjunctive subgoals (for example, to paint the ceiling, get paint and either a ladder or a table) as well as conjunctive subgoals. After the plan has been expanded to a certain depth, it may become evident that one disjunct is superior to the others by the nature of its interaction with the other actions in the plan. The Optimize Disjuncts critic builds and examines alternative TOMEs for each choice among disjuncts (or each combination of choices, if there is more than one disjunction). If one choice is found

---

*In general, determining what steps are appropriate is very difficult, and requires a full understanding of the SOUP code. The current implementation can only deal with variables that were bound by an explicit data retrieval statement.

to substantially minimize the number of actions in the overall plan, it is selected and the rest of the disjunction is ignored. The measure of the number of actions is crude, and for more careful implementations a measure with a more semantic basis (e.g., estimating the difficulty or cost of the overall plan) would be preferred.

3. Task-Specific Critics

In addition to the general-purpose critics described above, the NOAH system allows user-specified critics as a part of the definition of a task domain. Just as it makes good sense to separate local constraints from global ones having to do with general problem solving issues, it also makes good sense to allow the specifier of a task domain to encode these classes of constraints separately. Let us examine a few examples of portions of knowledge about a domain that would be difficult to express or deal with if they could not be expressed as critics. Examples are presented here from the CBC domain.

a. Tool Gathering -- Many of the operations from the CBC domain require the use of tools. Each operation that requires tools contains code that specifies their acquisition. But it makes global sense to acquire a number of tools in a single trip to the tool box. So a task-specific critic can be defined to reorder the steps of the plan to aggregate the tool gathering operations.

b. Limitations of an Apprentice -- The human apprentice who is diagnosing and repairing equipment can only be asked to use two hands, and can only be asked to be in one place at one time. These constraints are not related to any particular actions, but are constraints on the overall plan that the apprentice is asked to carry out.

For example, at one level of detail the apprentice may be asked to "Connect the cover to the frame." In more detail, he might be asked to "Position the cover on the frame, and then attach the cover to the frame." At this level of detail, the plan is fine as stated. But if each step of this plan is expanded, the resulting plan would be "Align the cover on the frame so that the cover is right-side-up and the screw holes line up. Then get a screwdriver, and screw in each of the screws." The plan at this level is ridiculous, since the act of aligning the cover will be undone when the screwdriver is fetched. So a critic that is based on a simple model of the physical limitations of a human would reorder this plan so that the screwdriver is fetched before the cover is aligned.

D. Using the Nonlinear Representation

We have just described in detail NOAH's approach to problem solving: the progressive expansion and criticism of nonlinear plans. This section will present a number of examples of blocks problems that NOAH's planner can solve. The first example will be presented in detail. The others will be displayed graphically, and only points of special interest will be discussed in the text.

A complete listing of the SOUP code used to define the semantics for this domain is presented in Appendix B.

1. Three Blocks

We are now ready to see how NOAH solves a simple problem. In the initial state, Block C is on Block A, and Block B is by itself. The goal is to achieve a new configuration: Block A is on Block B, and Block B is on Block C, as shown in Figure 9.

Initial State:
(ON C A)
(CLEARTOP B)
(CLEARTOP C)

Goal State:
(AND(ON A B)
(ON B C))
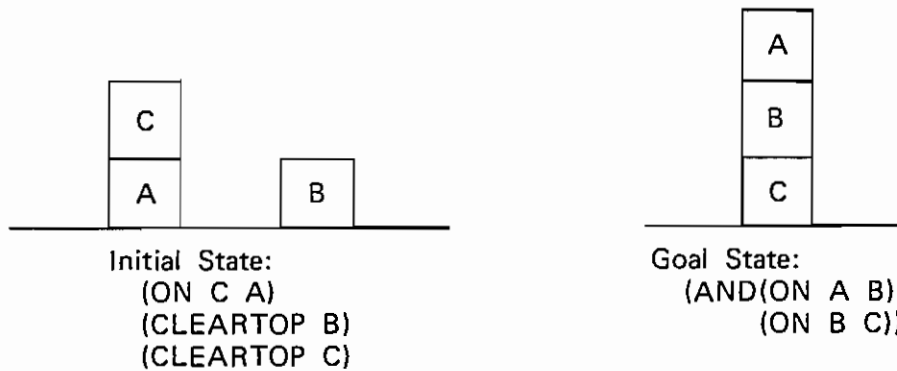
FIGURE 9   EXAMPLE PROBLEM

The initial state is expressed to the system as a set of QLISP assertions:

    (ON C A)
    (CLEARTOP B)
    (CLEARTOP C).

NOAH is invoked with the goal: (AND (ON A B) (ON B C)).

The system builds an initial procedural net that consists of a single GOAL node. The node is to achieve the given goal; its body is a list of the task-specific SOUP functions, in this case CLEAR and PUTON. It then applies the planning algorithm to this one-step plan, which is depicted in Figure 10a. The conjunction is split up, so that each of its conjuncts is achieved independently. PUTON is the relevant function for achieving both conjuncts, but the system does not immediately invoke PUTON. Rather, the system builds a new GOAL node in the procedural net to represent each invocation. The nodes are to achieve (ON A B) or (ON B C), and have PUTON as their body. The original plan has now been completely expanded to a greater level of detail, and so the critics are applied. At this level, they find no problems with the plan that was generated. The new plan is shown in Figure 10b.

FIGURE 10   PLANS FOR EXAMPLE PROBLEM

48
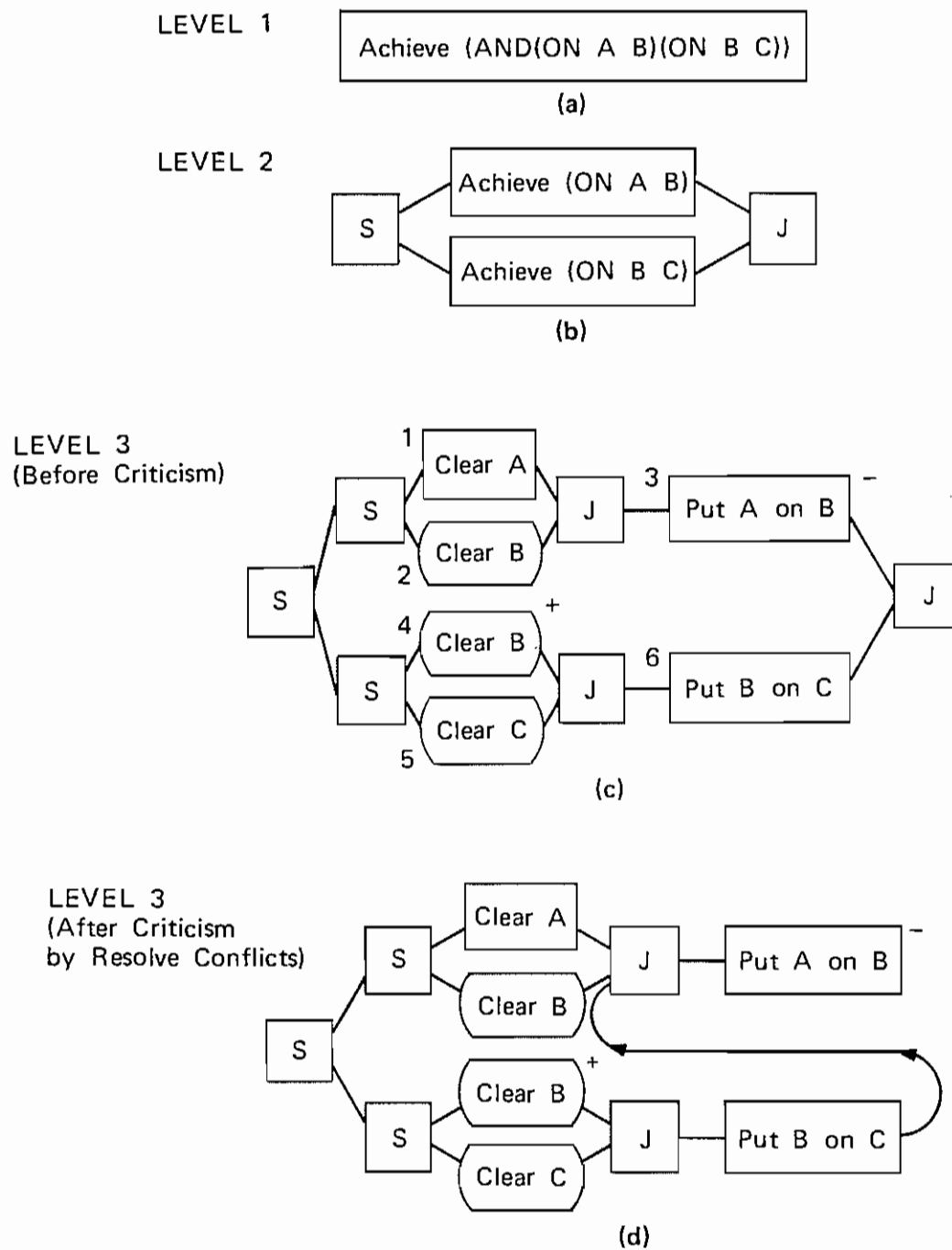
LEVEL 3
(After all Criticism)
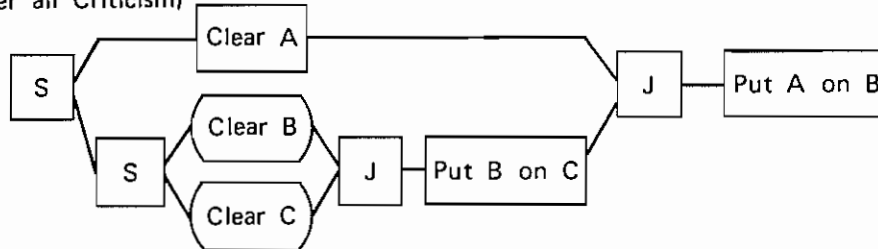
(e)

LEVEL 4
(Before Criticism)
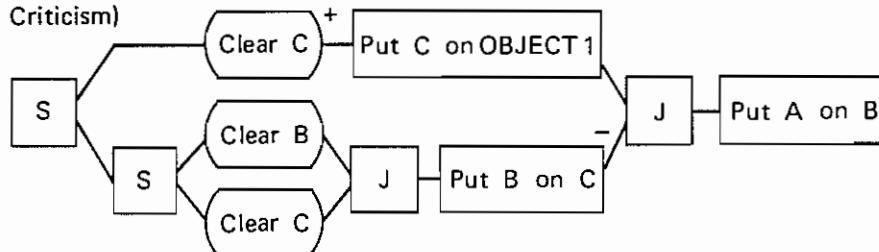
(f)

LEVEL 4
(After Criticism
by Resolve Conflicts)
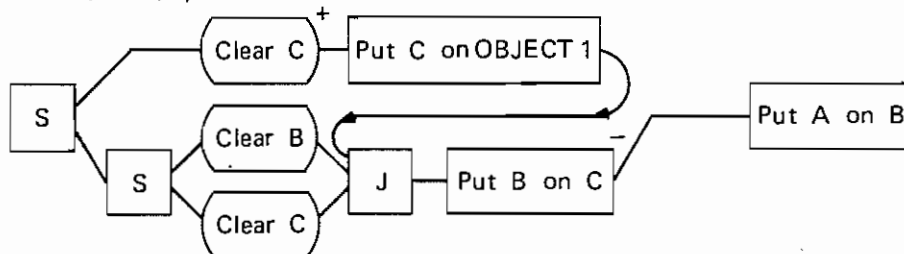
(g)

LEVEL 4
(After all Criticism)

(h)
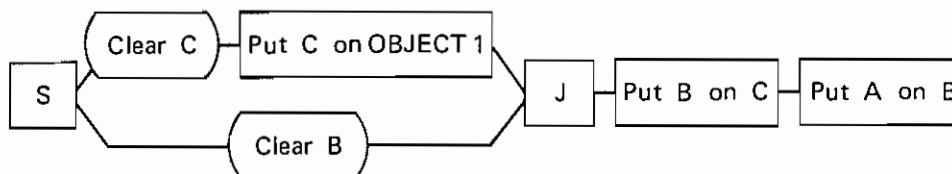
FIGURE 10   PLANS FOR EXAMPLE PROBLEM   (Concluded)

The new plan is now expanded. When the GOAL nodes for achieving (ON A B) and (ON B C) are simulated, PUTON is applied to each goal expression. PUTON causes the generation of a new level of GOAL nodes. When the entire plan has been expanded, the resulting new plan appears as in Figure 10c. The nodes of the plan are numbered to aid in explaining the actions of the critics.

The critics are now applied to the new plan. Resolve Conflicts generates a TOME, a table of all the expressions that were asserted or denied more than once during the expansion. The table is shown in Figure 11a. This table is then pared down by eliminating from consideration those preconditions that are denied by their own purposes. For example, (CLEARTOP C) is a precondition for the purpose (ON B C), so it is not a conflict that achieving (ON B C) at Node 6 makes (CLEARTOP C) false. Now, any expression for which there is only a single remaining effect is removed from the table. The resulting table, shown in Figure 11b, displays all the conflicts created by the assumption of nonlinearity.

FIGURE 11

TABLE OF MULTIPLE EFFECTS FOR EXAMPLE PROBLEM

(Node numbers refer to Figure 10c.)

11a – Original Table

CLEARTOP B: Asserted – Node 2 ("Clear B")
                     Denied – Node 3 ("Put A on B")
                     Asserted – Node 4 ("Clear B")

CLEARTOP C: Asserted – Node 5 ("Clear C")
                     Denied – Node 6 ("Put B on C")

11b – Refined table

CLEARTOP B: Denied – Node 3 ("Put A on B")
                     Asserted – Node 4 ("Clear B")

Resolve Conflicts now resolves the conflict by reordering the plan to place the endangered purpose [the node achieving (ON B C)] before the violating step [the node achieving (ON A B)]. The transformed plan is shown in Figure 10d.

Since no formal objects were generated at this level of detail, Use Existing Objects does not transform the plan further. Eliminate Redundant Preconditions is now applied, and the resulting plan is shown in Figure 10e. Note that the major restriction in the solution to the problem, that B must be placed on C before A is placed on B, has been incorporated into the plan. This has been accomplished directly, constructively, and without backtracking.

The critics having been applied, the system expands the new plan. This results in the generation of a new, yet more detailed plan, shown in Figure 10f. The critics are then applied. An analysis similar to that described above enables Resolve Conflicts to discover that (CLEARTOP C) might be violated when achieving (ON B C). Thus, the plan is rearranged, as shown in Figure 10g, so that (ON C Object1), the endangered purpose, is achieved before (ON B C).

Use Existing Objects again finds no formal objects that can be unified with existing ones. After Eliminate Redundant Preconditions cleans up the plan, it appears as in Figure 8h. The final plan is: Put C on Object1; Put B on C; Put A on B. Essentially, the plan is now completely linearized. The planning system has chosen the correct ordering for the subgoals, without backtracking or wasted computation. By avoiding a premature commitment to a linear plan, the system never had to undo a random choice made on the basis of insufficient information.

2. Four Blocks

The solution to this problem illustrates the use of the "Use Existing Objects" critic.

```
┌───┐   ┌───┐
│ C │   │ D │
├───┤   ├───┤
│ A │   │ B │
└───┴───┴───┘
```

Initial State:
   (ON C A)
   (ON D B)
   (CLEARTOP C)
   (CLEARTOP D)

```
┌───┐
│ A │
├───┤
│ B │
├───┤
│ C │
├───┤
│ D │
└───┘
```

Goal State:
   (AND(ON A B)
     (ON B C)
     (ON C D))
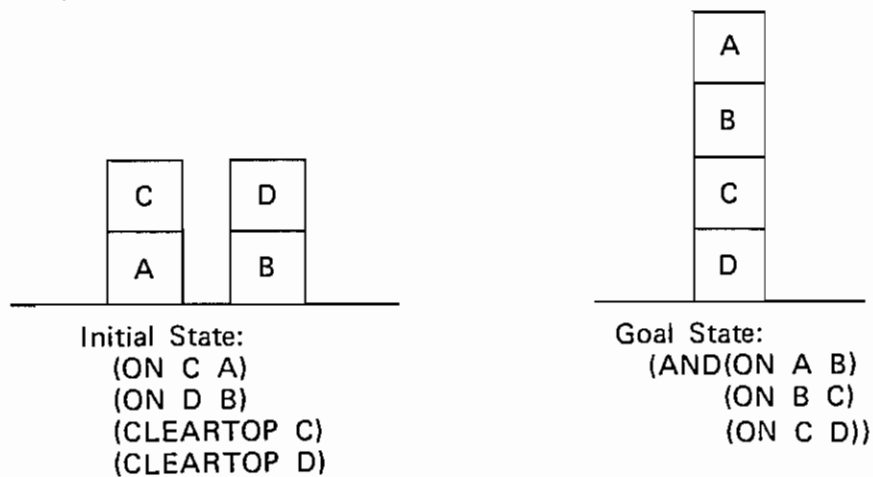
LEVEL 1 | Achieve (AND(ON A B)(ON B C)(ON C D)) |

The conjunctive goal is split into parallel goals.

LEVEL 2   S — Achieve (ON A B) — Achieve (ON B C) — Achieve (ON C D) — J
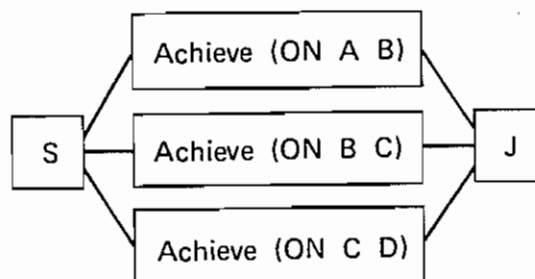
LEVEL 3
(Before Criticism)
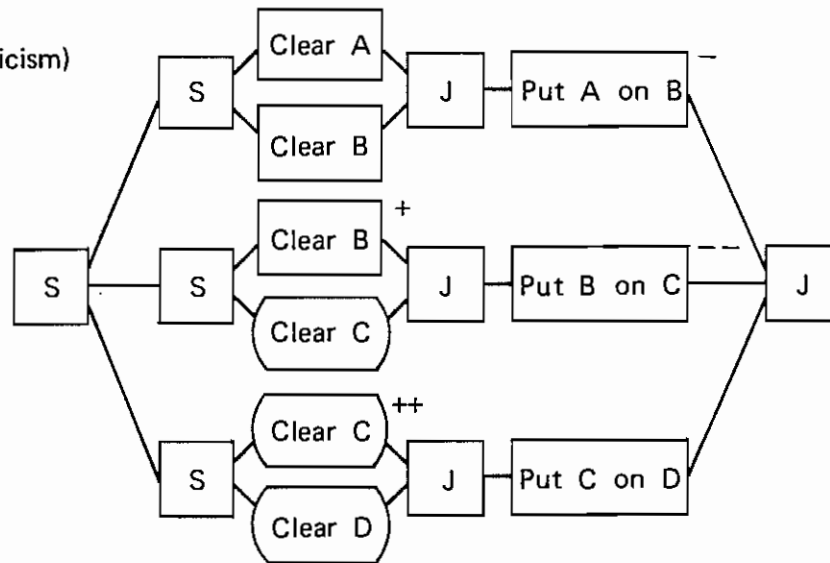
Resolve Conflicts notices two cases of a precondition (+ and ++) negated by parallel operations (- and --, respectively).

LEVEL 3
(After Criticism
by Resolve Conflicts)

Eliminate Redundant Preconditions cleans up the plan.

54
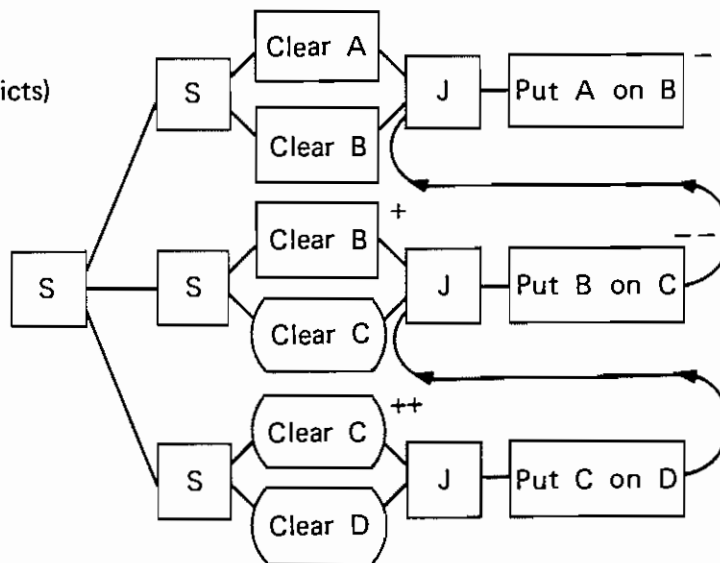
LEVEL 3
(After all Criticism)
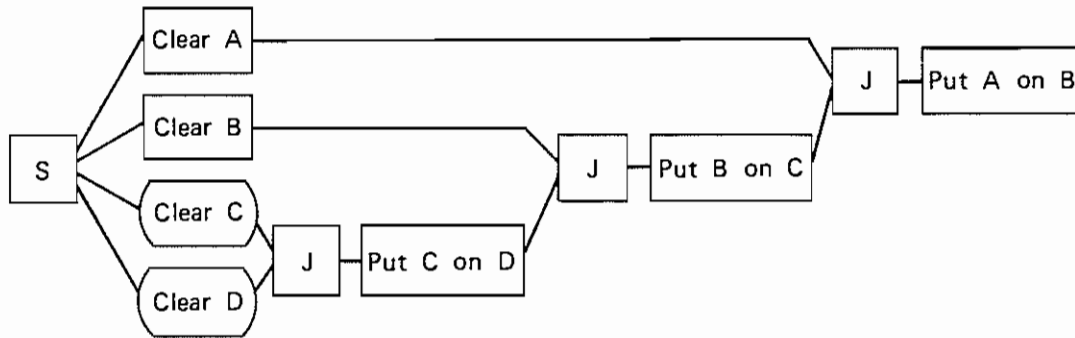
LEVEL 4
(Before Criticism)
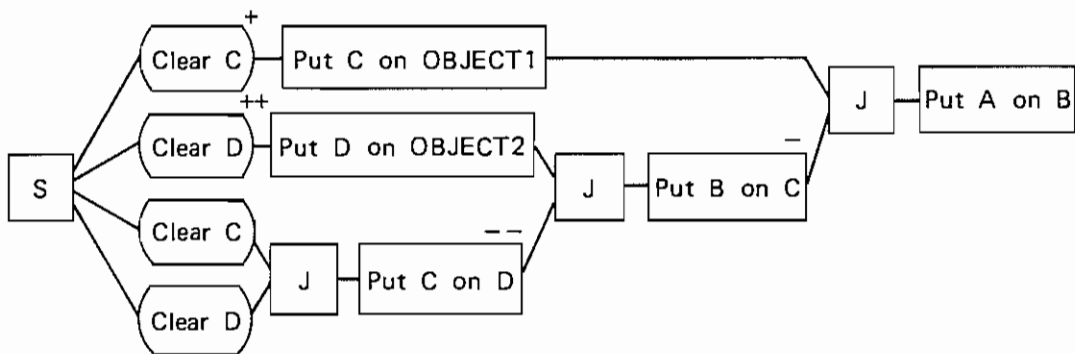
LEVEL 4
(After Criticism
by Resolve Conflicts)
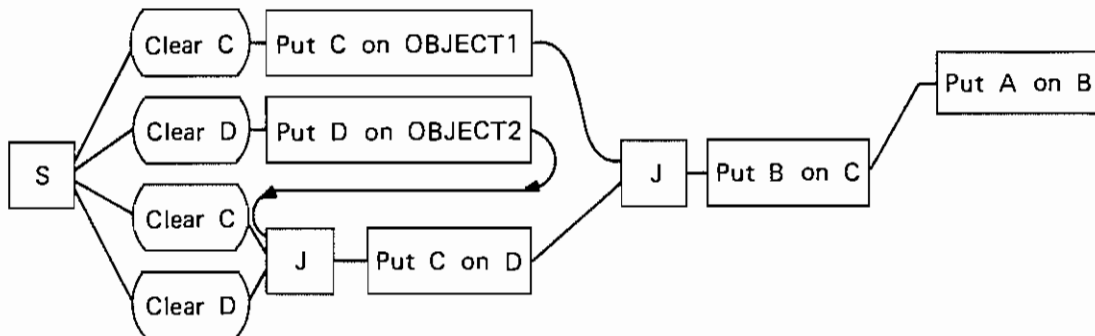
Use Existing Objects notices that the plan can be simplified by unifying the formal object, Object1, with Block D. The nodes that refer to putting C on D and on Object1 are merged.

LEVEL 4
(After Criticism
by Use Existing Objects)
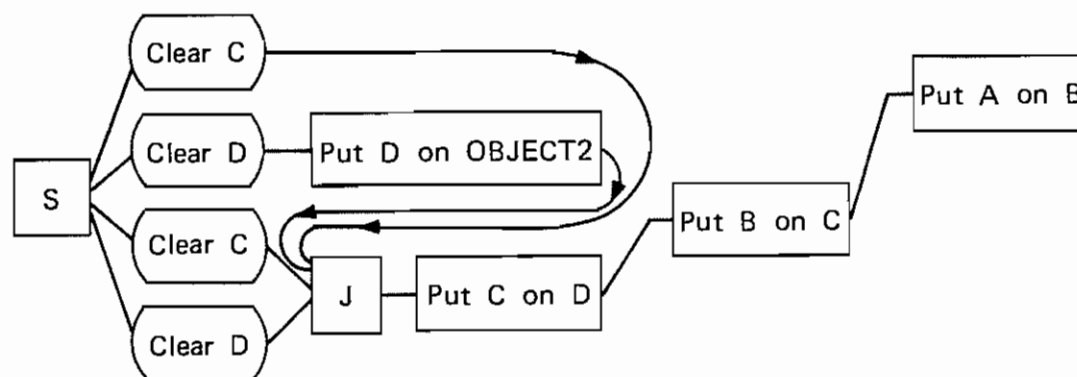
LEVEL 4
(After all
Criticism)
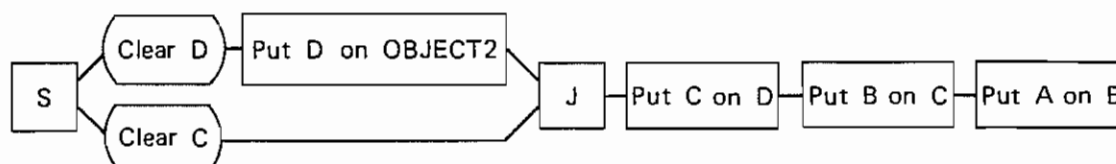
The final plan is: Put D on Object2, Put C on D, Put B on C, Put A on B.

3. Creative Destruction

This problem can only be solved by undoing a subgoal that is already achieved.

Initial State:
(ON A B)
(CLEARTOP A)
(CLEARTOP C)

Goal State:
(AND(ON A B)
(ON B C))

LEVEL 1   Achieve (AND(ON A B)(ON B C))
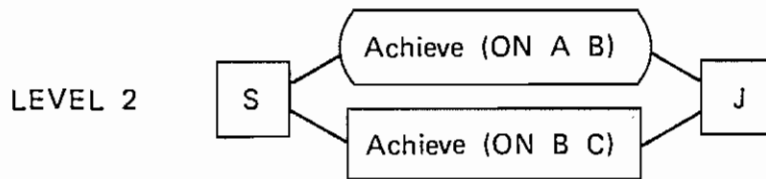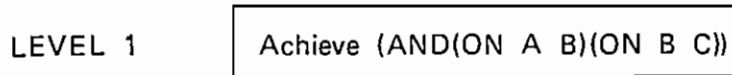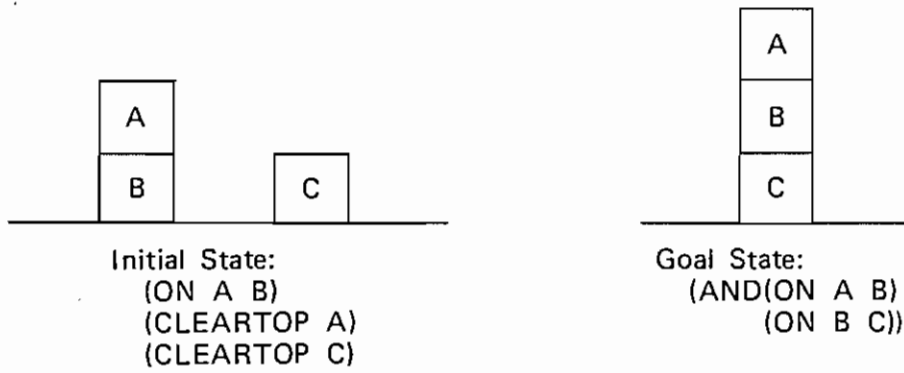
LEVEL 2

The first conjunct is a PHANTOM goal, since it is already true in the initial world model.

LEVEL 3
(Before Criticism)
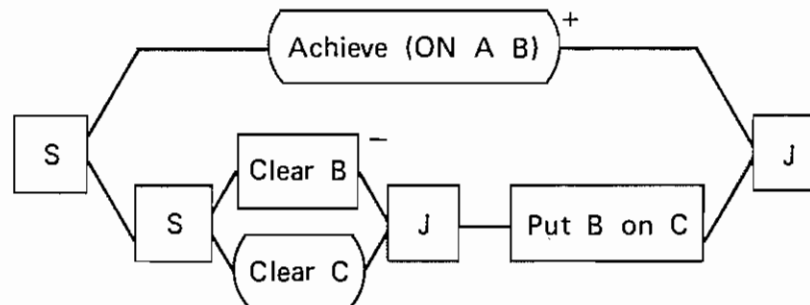
Resolve Conflicts notices that one node (−) deletes a precondition for a subsequent purpose. The precondition in this case is (ON A B), and the purpose is the initial conjunctive goal. The system therefore alters the PHANTOM goal (+) to become a genuine goal, to be achieved in time for the subsequent purpose.

LEVEL 3
(After Criticism)
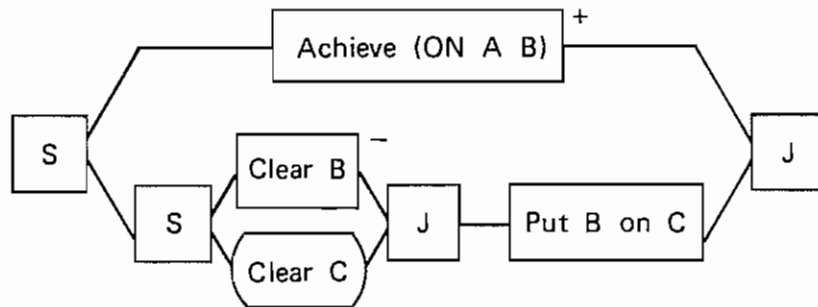
LEVEL 4
(Before Criticism)
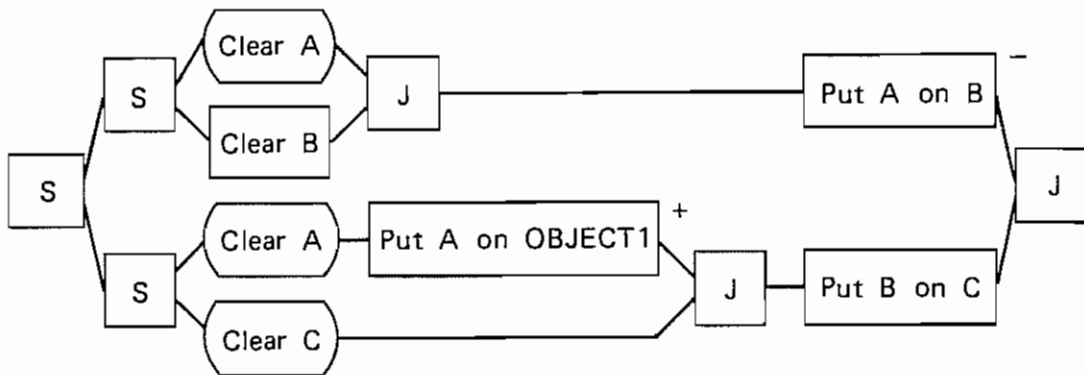
Resolve Conflicts notices that (CLEARTOP B) is asserted by one node(+) and deleted by another(−). It therefore reorders the plan.

58

LEVEL 4
(After Criticism
by Resolve Conflicts)

```
                  ┌─────────┐
              ┌──▶(Clear A )──┐         ┌───┐                              ┌───────────┐  ─
              │   └─────────┘  │    ┌───┤ J │────────────────────────────│ Put A on B│
          ┌───┤  ┌─────────┐   └───│    └───┘                            └───────────┘
          │ S │  │ Clear B │       │      │ ◀──────────────────────────────┐
          │   │  └─────────┘       │      ▼                                │
  ┌───┐   └───┘                    │                                       │
  │ S │                            │                                       │
  └───┘   ┌───┐  ┌─────────┐   ┌────────────────────┐ +                    │
      └───┤ S │─(Clear A )──│ Put A on OBJECT1    │      ┌───┐  ┌───────────┐
          │   │  └─────────┘  └────────────────────┘     │ J │──│ Put B on C│──┘
          └───┤  ┌─────────┐                             └───┘  └───────────┘
              └─(Clear C )───────────────────────────────┘
                 └─────────┘
```

Eliminate Redundant Preconditions cleans up the plan.

LEVEL 4
(After Criticism)

```
         ┌──────────┐ ┌────────────────────┐
     ┌──(  Clear A  )─│ Put A on OBJECT1    │──┐  ┌───┐ ┌───────────┐ ┌───────────┐
 ┌───┤   └──────────┘ └────────────────────┘  │  │ J │─│ Put B on C│─│ Put A on B│
 │ S │                                         │  └───┘ └───────────┘ └───────────┘
 └───┤   ┌──────────┐                          │
     └──(  Clear C  )──────────────────────────┘
         └──────────┘
```
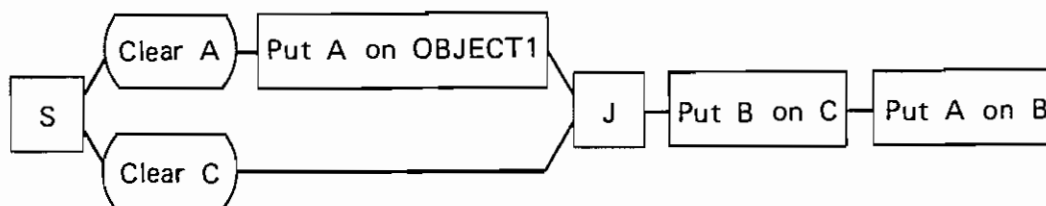
The final plan is: Put A on Object1, Put B on C, Put A on B.

4. Lumpy Blocks: Dealing with Disjunctions

This problem is designed to show a simple case of disjunctive optimization.

Initial State:
(CLEARTOP A)
(CLEARTOP B)
(CLEARTOP C)

Final State:
(AND(ON A B)
(STABLE C))
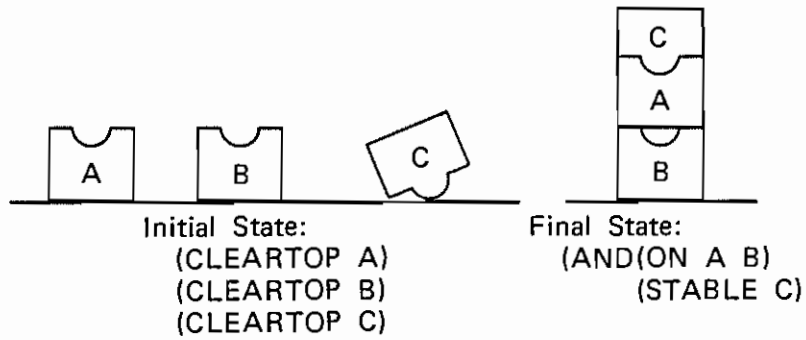
LEVEL 1 | Achieve (AND(ON A B)(STABLE C))

The conjunctive goal is split into parallel goals.

LEVEL 2

Achieve (ON A B)

S

Achieve (STABLE C)

J
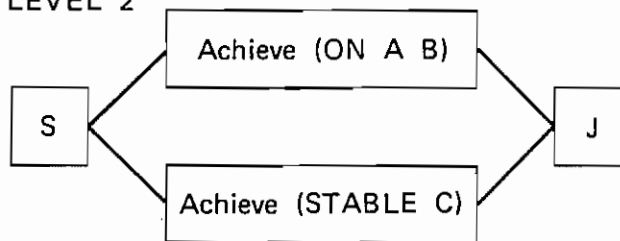
The disjunctive subgoal is split up.

60
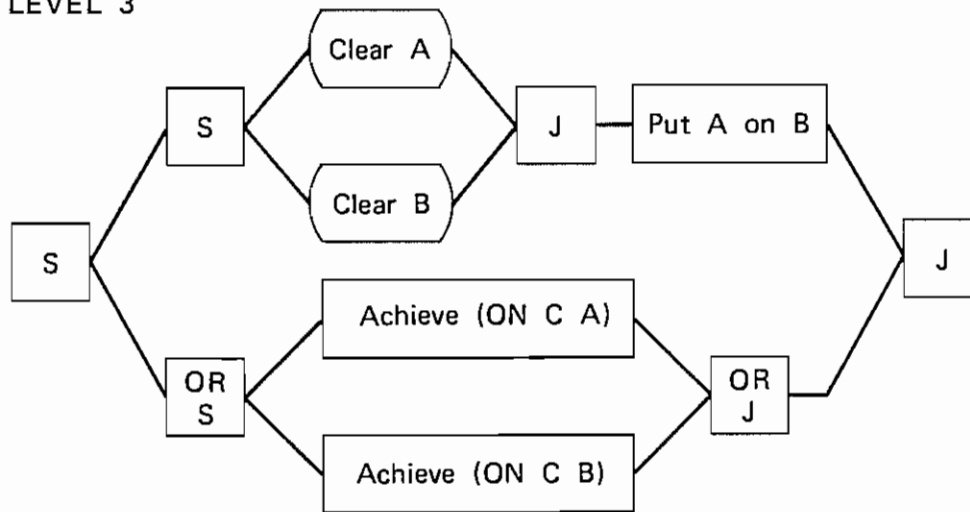
LEVEL 3

LEVEL 4
(Before Criticism)
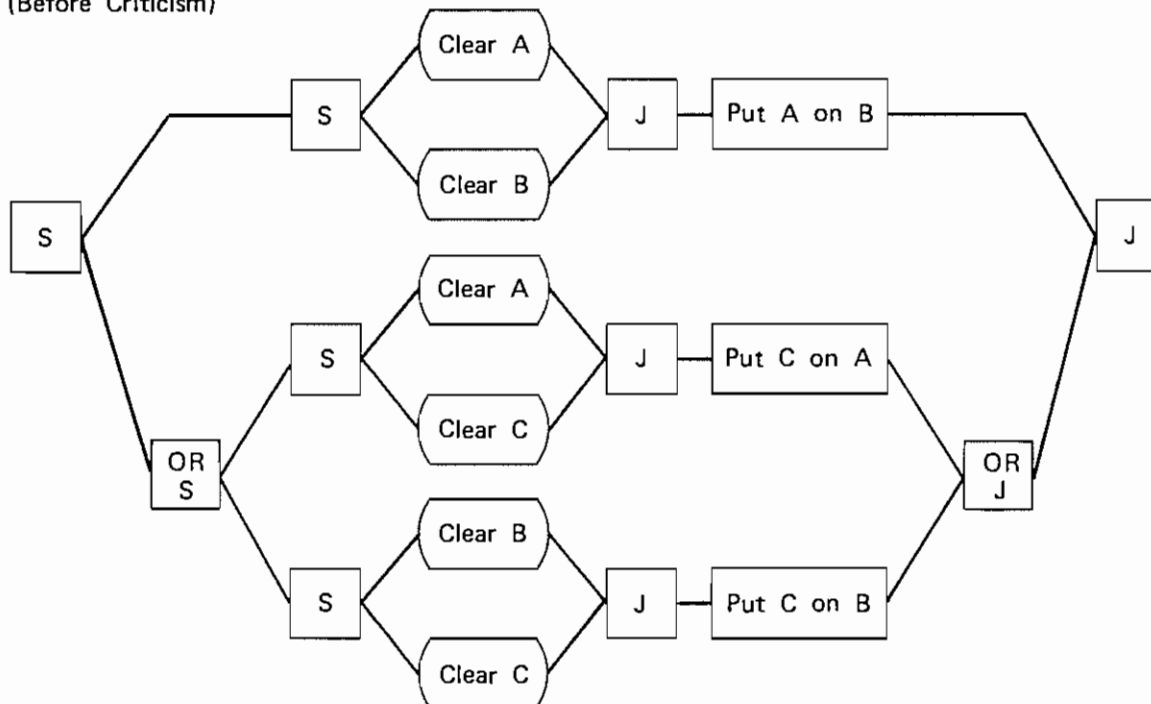
The TOME for the "Put C on A" disjunct contains two entries concerning "CLEARTOP B." The TOME for the "Put C on B" disjunct contains three entries for this expression. So, on a purely syntactic basis, the former disjunct is chosen. The system has no real understanding that the latter choice would require undoing a needed precondition.

LEVEL 4
(After Criticism by "Optimize Disjuncts"
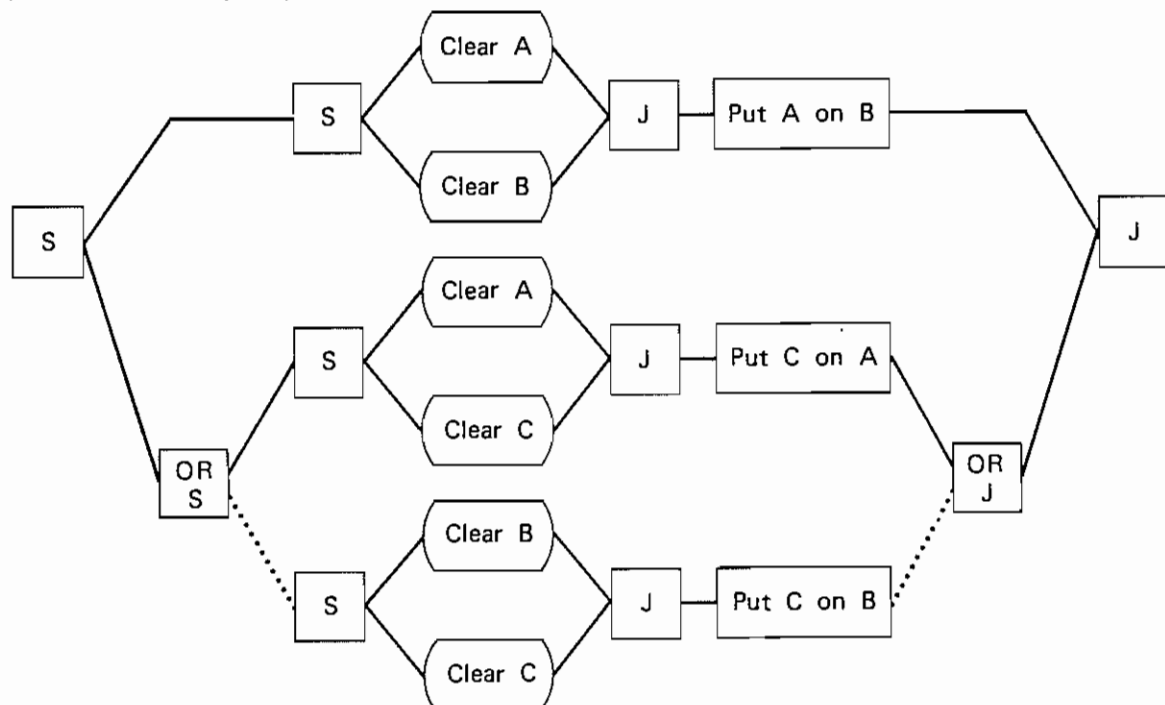
The other critics update this plan as before.

62

LEVEL 4
(After All Criticism)
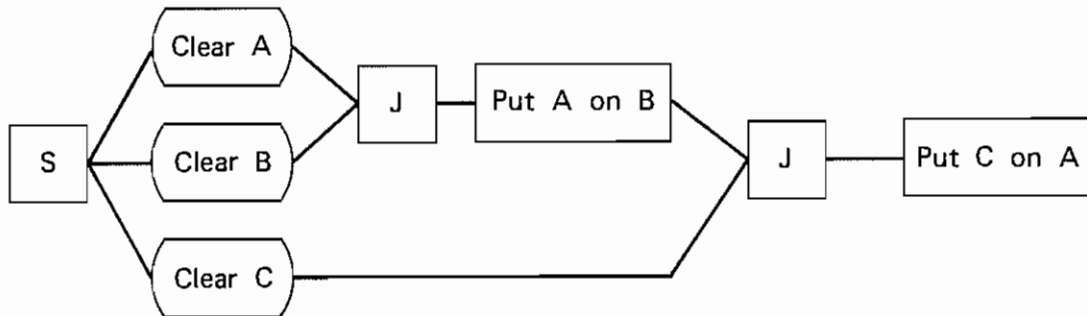
The final plan is: Put A on B; Put C on A.

5. Swapping Blocks: Nonlinearizable Interactions

This problem is a blocks world equivalent of the problem of interchanging the contents of two registers.
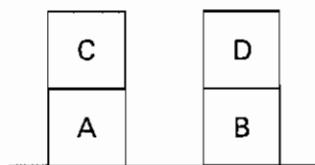
Initial State:
  (ON C A)
  (ON D B)
  (CLEARTOP C)
  (CLEARTOP D)
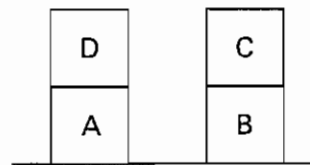
Final State:
  (AND(ABOVE D A)
    (ABOVE C B))

TA-740522-42

LEVEL 1

Achieve
(AND(ABOVE D A)(ABOVE C B))

LEVEL 2
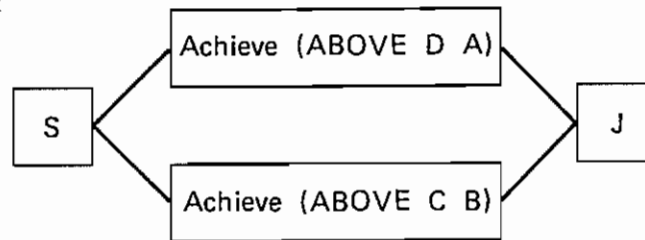
TA-740522-44

LEVEL 3
(Before Criticism)
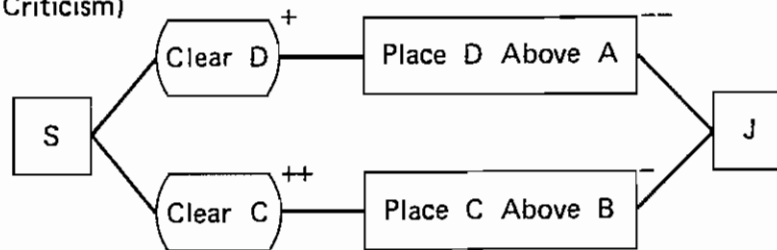
The purpose of each branch negates a precondition in the other branch. Thus, no simple linearization of the plan will eliminate the conflict. The Resolve Double Cross critic attempts to make each branch innocuous to the other. The critic determines, by examining a history of variable bindings, that the reason the "Place D above A" step caused "CLEARTOP C" to be deleted was that the variable ←TOP was bound to C when the query (PIS (ON A ←TOP)) was executed. The conflict would not occur if the PIS statement had returned a different value. So the critic inserts a new step in the plan before the point where the PIS would be executed. A similar operation is performed for the other branch. The rest of the plan is expanded again, so that the new expansion reflects the effects of the newly inserted step.

LEVEL 3
(After Criticism)
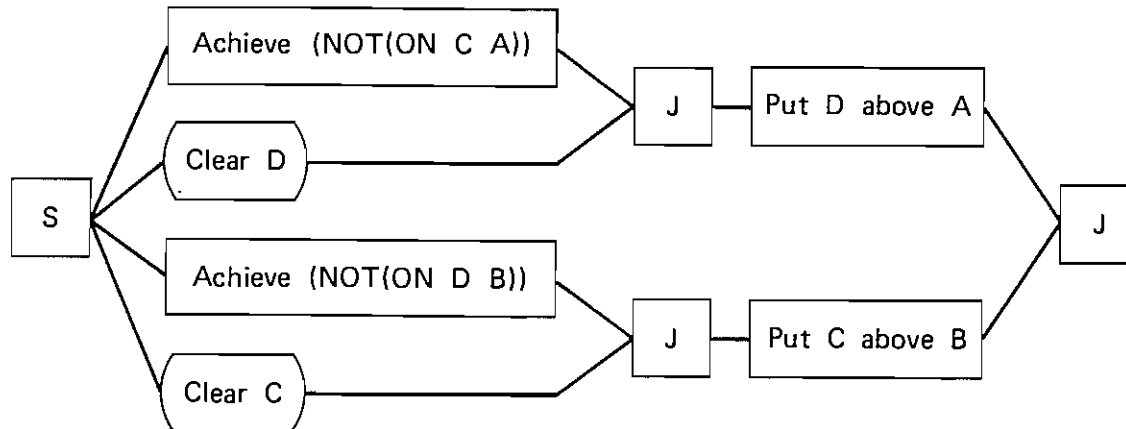
LEVEL 4
(After Criticism)
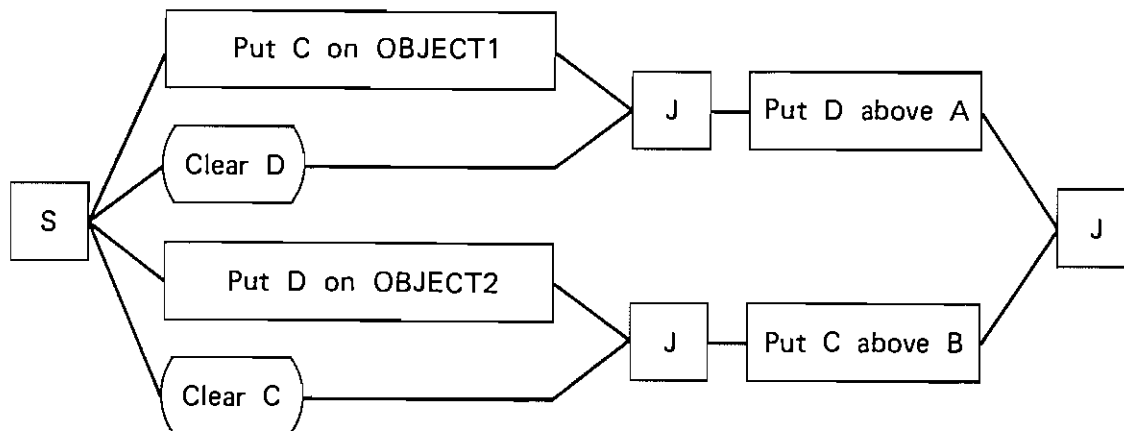
The plan, in its nonlinear form, is now completed. There is no reason to choose, at planning time, whether to work on putting D above A, or C above B first. However, once an arbitrary committment to an ordering is made during execution,

the system can further simplify the plan*. Let us suppose "Put C on Object1" was chosen. After it is executed, the portion of the plan that remains to be executed is as shown below.

LEVEL 4
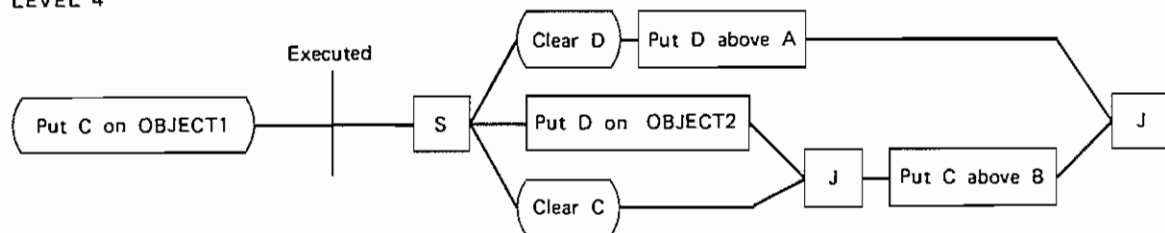
Execution may be thought of as a process of pulling the nonlinear plan through a hole that can only accept one step at a time. As each step is pulled through, the remaining plan is deformed into a new structure. The new form may be criticized just as during planning. In this case, the new structure is one which the Use Existing Objects critic can transform into a simpler one.

LEVEL 4
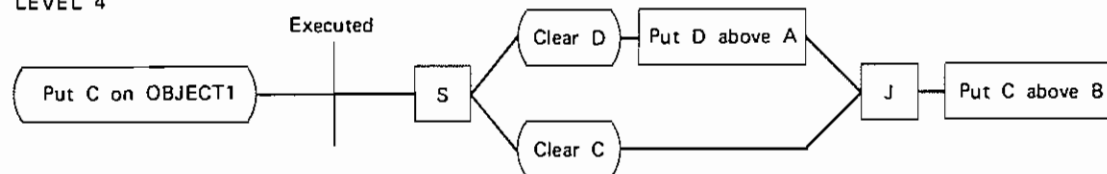
The rest of execution proceeds normally. The plan as executed is: Put C on Object1, Put D above A, Put C above B.

---

*The process of execution-time criticism has not been implemented in the current version of NOAH.

E. Using the Hierarchical Representation

1. Introduction

Thus far, all the examples that have been presented have had the property that the higher levels of planning successfully narrowed the solution space to include the solution. Now we shall present an example in which this is not the case, and some examples in which the system deliberately avoids narrowing the solution space.

2. Hierarchical Kernels

NOAH restricts the solution space on the basis of a cheap and nonexhaustive analysis of high-level constraints. Therefore, it must have a mechanism for checking as more detailed analyses are performed that the solution is reachable within the restricted solution space. There is a danger that the detailed expansion of a preceding action will obliterate a higher level precondition of a subsequent action. Furthermore, there is no way to know if a problem like this will arise until the expansion is actually carried out. For example, consider a pair of plans to set up experiments to test a monkey's problem solving behavior. One plan, "Acquire bananas, Put bananas on floor, Put box in corner," can be expanded to more detail with no difficulty. But consider a very similar plan: "Acquire bananas, Hang bananas on hook, Put box in corner." There is little to distinguish the two plans at the level of detail that they have been presented. But the second one cannot successfully be expanded by expanding each step individually. The expansion of the "Hang bananas" subplan will create the subplan: "Move the box under the hook,

Place bananas on hook." A further expansion will include setting down the bananas in order to push the box. This will enable the box to be successfully moved, but it will leave things in a state such that the subplan to place the bananas on the hook will no longer work (since the bananas are no longer in hand). Conflicts of this sort can be spotted by checking that the higher level plans are all being carried out successfully within the evolving plan at the current level.

In fact, the higher level plans do not need to be checked in their entirety. The system needs to check only a subset of the higher level actions as each node is being expanded. We shall call this subset the hierarchical kernel of the node being expanded. The check of the kernel is done by checking the effects of the preceding siblings of each ancestor node of the one currently being expanded. The preceding steps at the top level are checked to make sure that their effects are true in the current world model. If they are, then the system is assured that the highest level plan is being carried through at the current level. That is, all the steps in the highest level plan, up to the current node's top-level ancestor, have been carried out at the current level.

Associated with the top-level ancestor is its expansion into a subplan, one of whose steps is also an ancestor of the current node. The steps in this subplan that precede the ancestor are checked to make sure that their effects are reflected in the bottom-level plan. If so, then the portion of the subplan preceding the ancestor has worked at the bottom level and so the ancestor at this level is still appropriate. The ancestor's subplan is checked in a similar way, and the operation proceeds recursively until the node about to be expanded is reached. If all the

elements of the hierarchical kernel are present in the current world model, then the current node may be expanded with confidence that the resulting subplan will be relevant for achieving each higher level plan.

Formally, then, for a given node, the kernel consists of the node's preceding siblings, and the preceding siblings of all its ancestors. The preceding siblings of a given node are all the fellow children of the node's parent that precede the given node in time. For example, Figure 12 shows the kernel of an action in the detailed expansion of a procedural net for the experimenter and bananas example.
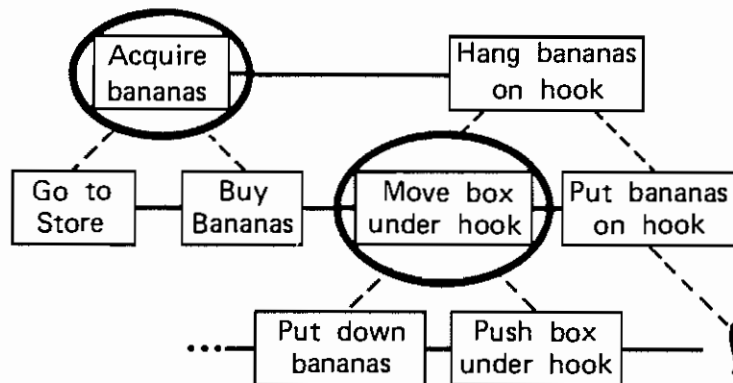
FIGURE 12   A KERNEL FOR THE "EXPERIMENTER AND BANANAS" PROBLEM

In the example, the problem solver is about to expand the "Put bananas on hook" node at the second level of the plan. Before doing so, it will check the node's kernel. The effects of the first node at the top level, namely that the experimenter has the bananas, is not true in the world model of the point in the new plan where the expansion will go. In the diagram, this point is indicated by the

exclamation mark. The check of the hierarchical kernel has thus revealed a flaw in the simple expansion of the plan.

When a check of the kernel reveals a problem, there are several alternative ways in which the problem solver can attempt to correct it. If there were arbitrary choices made at previous points in the expansion (for instance, the choice of one function in a GOAL node's body when others would also be applicable), then the alternatives to those choices could be explored. In our example, the experimenter might be able to call upon his very tall assistant to place the bananas on the hook. Another way to correct the plan is simply to plan to reestablish the purpose of the node whose effects are no longer preserved. In our example, this corresponds to inserting a new GOAL node at the exclamation point to achieve "On hook bananas." The current NOAH system uses this second approach, simply because it works on the problem to be presented below. A more sophisticated system would require a level of control to integrate attempts to solve the problem using both approaches, and other approaches as well.

The use of kernels to ensure that a detailed plan is still on the path sketched out at a higher level can be viewed as the program-generation analogue of the concept of "hierarchical debugging" suggested by Goldstein [14] for a program-understanding task.

A successful check of the hierarchical kernel does not guarantee that a problem will not arise. (The kinds of interaction that will be missed are discussed in Section A of Chapter V.) It is too expensive computationally to check every predicate required by every action. This was a major failing of the STRIPS-PLANEX

system. The kernel checking described here is an attempt to use the structure of the procedural net to perform a minimal set of checks that will catch most unwanted interactions. This section does not propose an approach to solving problems that is expected to be generally correct and applicable. Rather, it is a simple demonstration of a kind of use to which the power of the procedural net can be put.

a. Using the Kernel: The "Keys and Boxes" Problem -- The "keys and boxes" problem has been proposed by Michie [23] as a "benchmark" problem for problem solving programs. The problem is as follows:

In a room there is a robot with no sensory ability whatsoever. Four places are defined in the room: BOX1, BOX2, TABLE, and DOOR. Outside the room there is one place: OUTSIDE.

At DOOR there is a non-empty pile of red objects.

At BOX1 or BOX2 (we don't know which) there is a non-empty pile of keys, all of which fit the door. We don't know what is at the other box.

TABLE is empty.

The robot has three possible actions:
(1) Pick up -- If the robot is holding something, this action has no effect. Otherwise, some object at the location will be in the robot's hand when this action is completed.
(2) Put down -- If the robot is not holding anything, this action has no effect. Otherwise, the object in the robot's hand is added to the pile at the current location of the robot.

(3) Go to X -- The robot's location becomes X. If X is OUTSIDE, there

must be a key in the pile at DOOR or this action has no effect.

The robot has no way to tell if any action had an effect or not.

Initially the robot is at an undetermined place in the room, and it is

unknown if anything is in its hand. Figure 13 suggests the initial configuration. The

problem is to develop a set of actions that will ensure that a red object is OUTSIDE.

Initial Situation:
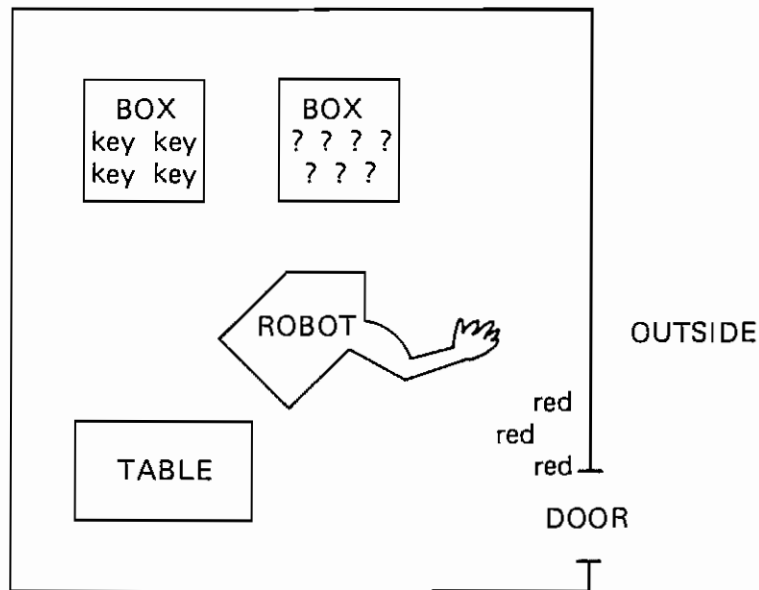
FIGURE 13   THE "KEYS AND BOXES" PROBLEM

The problem as specified here is actually considerably harder than that

axiomatized by Michie. In his specification of the problem, if an object is in the

robot's hand initially, it is of the same type as the other objects at the place where

the robot is standing. The above specification allows the object to be of any type,

and the corresponding solution requires four more steps.

The solution requires 21 steps. The reader is encouraged to attempt the problem before reading further. It is not trivial. About half of a class of graduate students in Computer Science could not solve it in ten minutes. The solution requires an ability to deal with a partially specified initial state, and with actions whose effects are only partially specified. It requires the integration of subplans that have strong interactions. And the solution to the problem has a rather large number of primitive steps.

This problem is very difficult for more primitive planners that rely solely on heuristic search. If a planner using pure breadth-first search would analyze one action every microsecond, it would require over 8000 years on the average to find a solution.

As with the previous examples, NOAH's operation will be presented graphically. The set of SOUP functions for this problem is shown in Appendix C. In addition to models of the primitive actions specified in the problem, the SOUP semantics specify higher level actions for transferring an object from one place to another, for transferring an object of a given type to a given place, and for emptying the robot's hand. NOAH's success on this problem stems from its ability to use these higher level actions.

The system is presented the goal (AT (RED ←R) OUTSIDE).

LEVEL 1

```
Achieve (AT (RED ← R) OUTSIDE
```

MOVEINSTANCE is the function whose pattern matches the goal. When it is evaluated, it determines that the robot must transfer a red object Object1 from the DOOR to OUTSIDE.

LEVEL 2

```
┌──────────────────────────────────────────────┐
│  Transfer OBJECT1 from DOOR to OUTSIDE         │
└──────────────────────────────────────────────┘
```

LEVEL 3

```
┌──────────────────┐   ┌──────────────────┐   ┌──────────────────┐
│  Acquire OBJECT1  │───│  Go to OUTSIDE    │───│  Put down OBJECT1 │
└──────────────────┘   └──────────────────┘   └──────────────────┘
```

To acquire Object1, the robot must be assured that its hand is empty. It must put down the possible object Object2 that it may be initially holding. Since the planner had no information about which place is best to put it down, a SOME node is inserted in the plan enumerating the possible places Object2 could be placed.

LEVEL 4

```
  ╭─────────────────╮   ┌────────┐   ┌────────┐   ┌────────────┐   ┌────────┐   ┌────────┐
 ╱ SOME              ╲  │ Go to  │   │ Pickup │   │ Bring a key│   │ Go to  │   │ Put down│
│  Put down OBJECT2    │─│ DOOR   │───│OBJECT1 │───│ to DOOR    │───│OUTSIDE │───│OBJECT1 │
│  in one of TABLE or  │ └────────┘   └────────┘   └────────────┘   └────────┘   └────────┘
 ╲ DOOR or BOX1       ╱
  ╲  or BOX2         ╱
   ╰───────────────╯
```
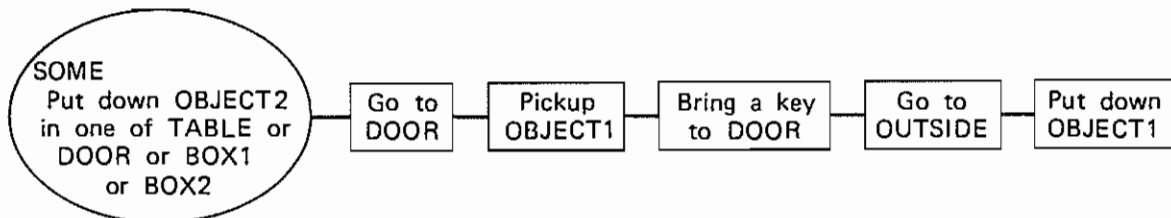
The planner first tries to place Object2 on the TABLE. The choices for the SOME node were ordered by the difficulty in retrieving a new object placed there. The TABLE was best since it was initially empty.

To bring a key to the door, the robot must empty its hand of Object1, the red object it acquired at the DOOR. Another SOME node is inserted in the plan, providing choices for the place to put Object1.

LEVEL 5

```
┌─────────┐  ┌──────────┐  ┌─────────┐  ┌─────────┐   ╱─────────────────╲
│ Go to   │──│ Put down │──│ Go to   │──│ Pick up │──│ SOME              │
│ TABLE   │  │ OBJECT 2 │  │ DOOR    │  │ OBJECT1 │  │ Put down OBJECT1   │
└─────────┘  └──────────┘  └─────────┘  └─────────┘  │ in one of TABLE or │
                                                      │ DOOR or BOX1       │
                                                      │ or BOX2            │
                                                      ╲─────────────────╱
```

```
      ┌──────────────────────┐  ┌─────────┐  ┌──────────┐
      │ Transfer objects from │──│ Go to   │──│ Put down │
      │ BOX1 and BOX2 to DOOR │  │ OUTSIDE │  │ OBJECT1  │
      └──────────────────────┘  └─────────┘  └──────────┘
```
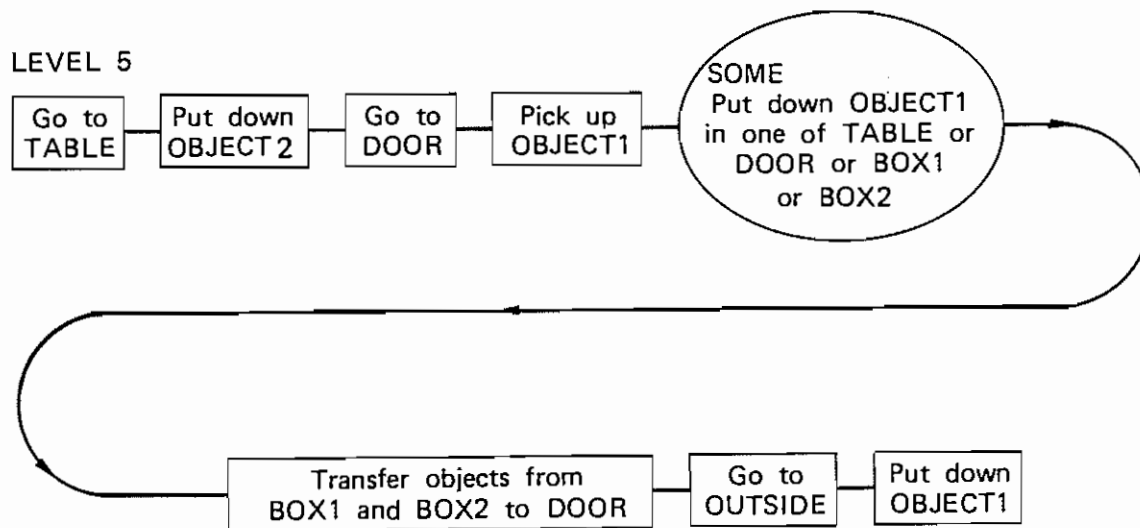
The planner first tries to put Object1 down at the TABLE, since it is still the least cluttered place. The action of transferring the key to the door is expanded into parallel subplans to transfer an object from each box to the door. By using a mechanism to be described in the following subsection, NOAH could handle an alternative problem that specified 541 boxes instead of two boxes, with very little additional effort*.

---

*John McCarthy brought this interesting variant of the problem to the author's attention.

LEVEL 6
(During expansion)

| Go to TABLE | Put down OBJECT2 | Go to DOOR | Pick up OBJECT1 | Go to TABLE | Put down OBJECT1 |

| S |

| Acquire OBJECT3 | Go to DOOR | Put down OBJECT3 |
| Acquire OBJECT4 | Go to DOOR | Put down OBJECT4 |

| J | ...
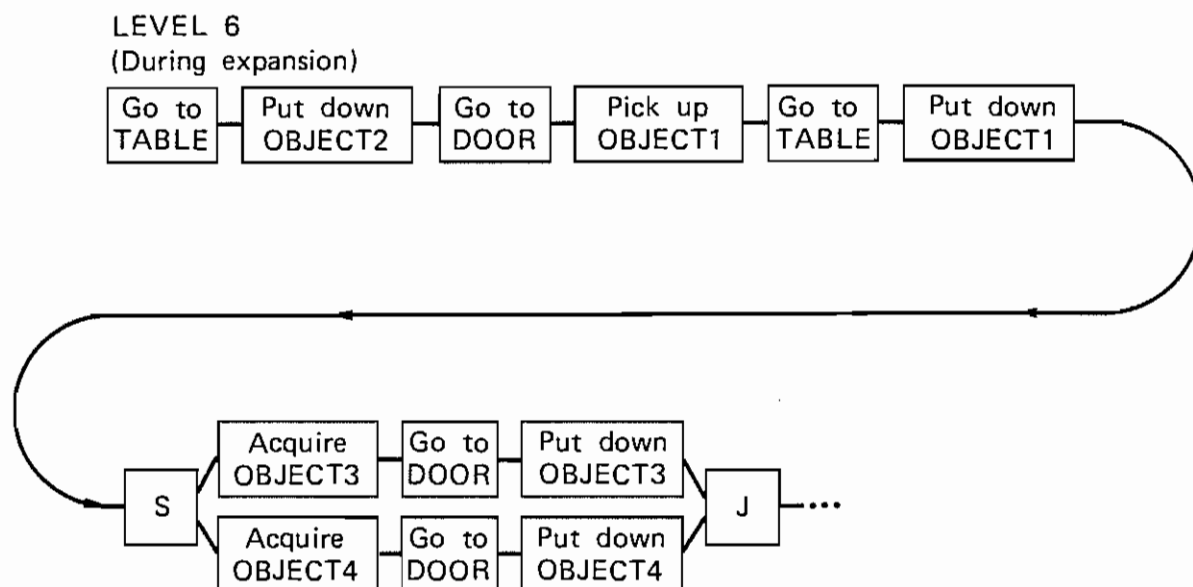
At this point, the check of the hierarchical kernel (which is shown in Figure 14) reveals that the robot is no longer holding Object1, and so there is no point in going OUTSIDE. The system now tries the alternatives that were set up at the two SOME nodes (resulting in 15 alternative expansions), two of which reach the same point where the kernel fails, and the rest of which fail at higher levels. At this point, the system decides that there is no way to avoid the kernel failure, and so the best that can be done is to alter the plan to reestablish the purpose whose precondition had been violated (in this case, the original goal (AT Object1 OUTSIDE)).

76
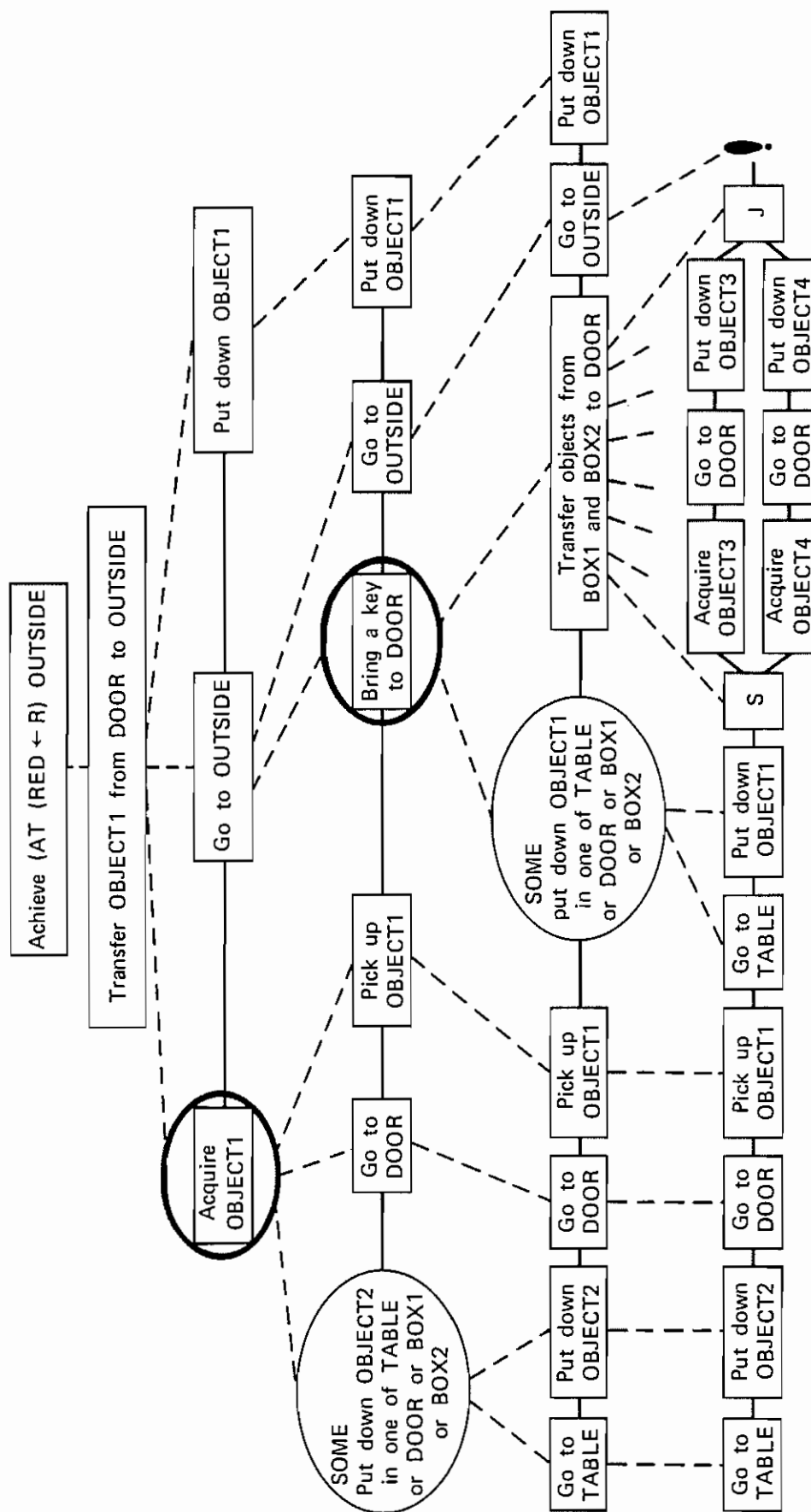
FIGURE 14   UNSATISFIED KERNEL IN "KEYS AND BOXES" PROBLEM

LEVEL 6
(After purpose of violated step is added as new goal)

| Go to TABLE | Put down OBJECT2 | Go to DOOR | Pick up OBJECT1 | Go to TABLE | Put down OBJECT1 |

S

| Acquire OBJECT3 | Go to DOOR | Put down OBJECT3 |
| Acquire OBJECT4 | Go to DOOR | Put down OBJECT4 |

J

| Achieve (AT OBJECT1 OUTSIDE) |
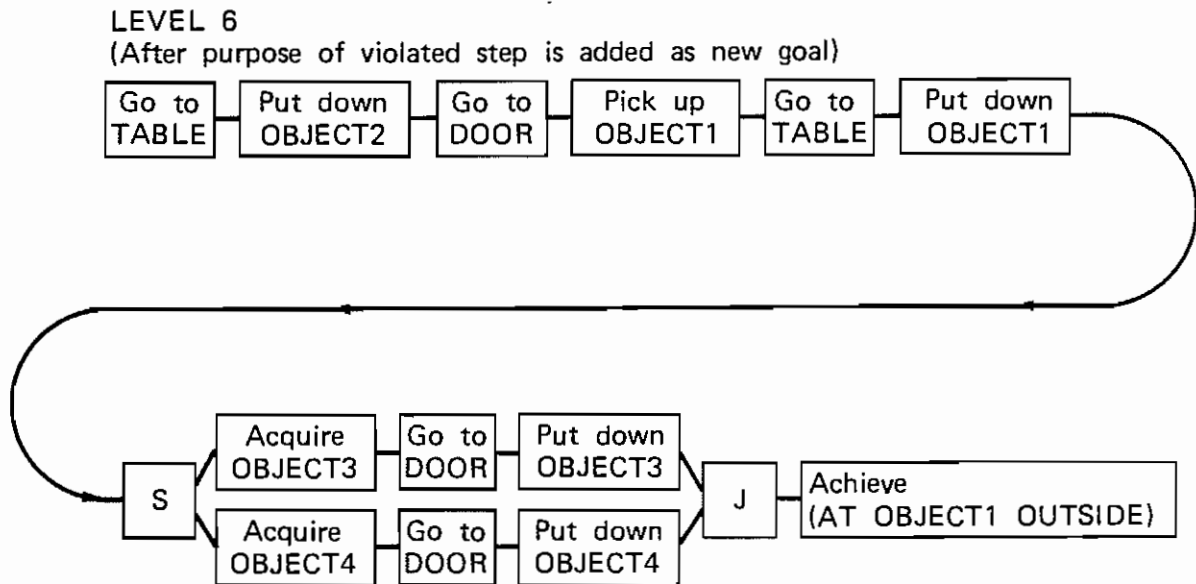
The altered plan is now expanded in the usual way.

The newly introduced goal can be achieved by transferring two objects from the table to OUTSIDE.

LEVEL 7

···· S

| Go to BOX1 | Pick up OBJECT3 | Go to DOOR | Put down OBJECT3 |
| Go to BOX2 | Pick up OBJECT4 | Go to DOOR | Put down OBJECT4 |

J

| Transfer objects from TABLE and TABLE to OUTSIDE |
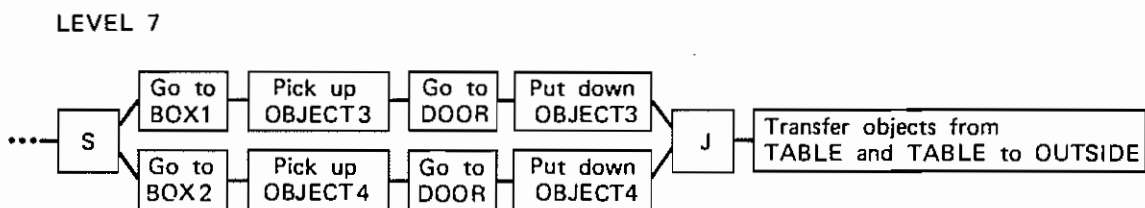
The transferring operation is broken down into two independent transfers. One of Object5 and Object6 is the original Object1; the other is the original Object2.

LEVEL 8
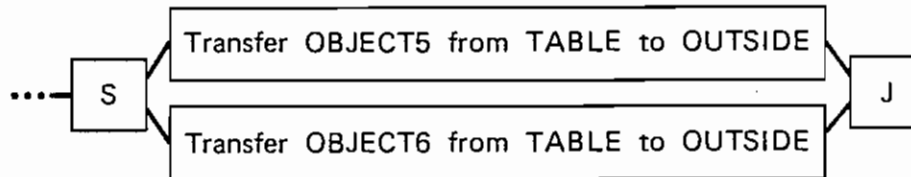
LEVEL 9
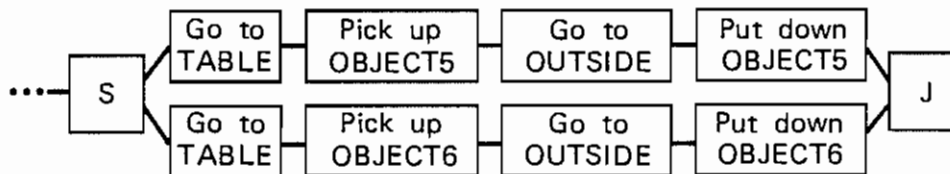
The final plan is: Go to TABLE, Put down, Go to DOOR, Pick up, Go to TABLE, Put down, Go to BOX1, Pick up, Go to DOOR, Put down, Go to BOX2, Pick up, Go to DOOR, Put down, Go to TABLE, Pick up, Go to OUTSIDE, Put down, Go to TABLE, Pick up, Go to OUTSIDE, Put down. (The final Put down operation is not required in the original statement of the problem, but is necessary in the axiomatization used here.)

The use of hierarchical kernels has much in common with Sussman's [34,35] approach of debugging almost-right plans. It is costly in comparison with the other planning mechanisms of NOAH, but the hierarchical planning strategy necessitates its use.

F. Planning for Iterative Actions

Although iterative actions are quite common in both everyday life and in computer programming, there have been very few attempts by problem solving systems to deal with them. Perhaps the most successful has been the automatic programming system of Buchanan and Luckham [5], though the system required a great deal of information to be specified about the iteration beforehand.

The SOUP language provides for the specification of two kinds of iterations, for building up a class of objects, and for iterating through a class of objects. These iterations are specified by PBUILD and PBREAK statements, respectively.

The structure of the procedural net allows an iteration to be modelled as a single action at a higher level of abstraction. For instance, an iteration might specify the tighening of each of four bolts. The iteration can be modelled at a higher level as a single action to bolt down a component.

NOAH does not need to expand out every cycle of an iteration. A mechanism has been developed that allows the planner to expand only one cycle fully, and then to model the effects of the others by analogy. When a BUILD or BREAK node is simulated, the code inside the ITERATE statement associated with the node is evaluated. This results in the creation of nodes modelling a single pass through the loop. Then a special REPLICATE node is added, in parallel with the expansion of the single pass. This node contains restart information for continuing the iteration. Also, if the expansion of the single pass caused any new formal objects to be generated, copies of them are created for each unexpanded pass around the loop. This is

because a number of actions in sequence may be loops that refer to the same objects, and a subsequent loop might be expanded out in greater detail than the initial loop. The system needs to understand where the objects came from so that it can model them and describe them to a user.

The subplan representing the single pass through the loop and the REPLICATE node are linked together with special LOOPSPLIT and LOOPJOIN nodes.

For example, let us look at the expansion of a step of a plan to screw a cover onto the housing of an air compressor belt. The expansion is shown in Figure 15. The single action "Screw Belt Housing Cover to Belt Housing Frame" is expanded into a sequence of three iterations. The iterations are represented by single nodes that model getting 10 screws, loosely fastening 10 screws, and tightening 10 screws. This sequence could be expanded into a 30-step plan, and we could brag about how the planner could handle really long plans, but that is not the intelligent thing to do. Instead, each iteration is expanded only once. REPLICATE nodes are used to take the place of all the other expansions. The REPLICATE node contains all the information necessary to expand the remaining iterations. The expansion is not actually done, however, unless it is needed to specify additional detailed actions during plan execution. We will come back to this example when we discuss execution monitoring, since the response of the user is what determines further expansions.
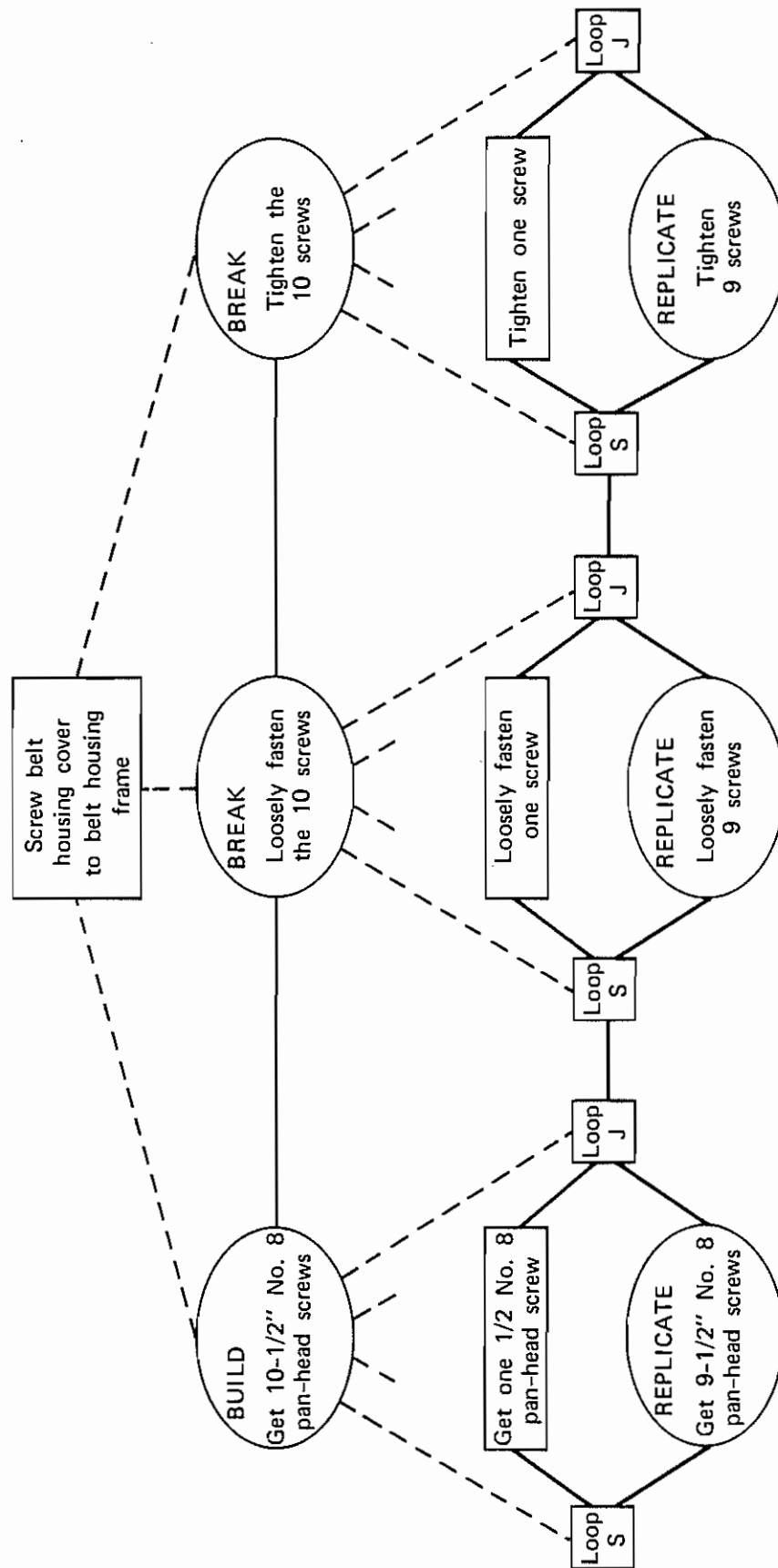
FIGURE 15   A SUBPLAN FOR INSTALLING THE BELT HOUSING COVER

G. Comparison with Other Current Work

We have seen how a variety of problems which can be represented as conjunctive goals have simple, straightforward solutions in NOAH. There are a number of other problem solving systems that use alternative approaches to solve similar problems. Among these are Sussman's use of debugging [34,35], Tate's search in a space of "tick lists" [38], and the approach of passing goals and actions backward over a partial plan, which is used by Waldinger [39] and Warren [40].

The approach presented in this paper is in many ways antithetical to that of Sussman's HACKER*. HACKER attacks conjunctive goals by making a "linear" assumption. That is, conjunctive goals are assumed to be independent and additive, and so to achieve the overall goal each conjunct may be achieved in sequence. The system is explicitly aware of this assumption. If the developing plan fails, it can be debugged by comparing the problem that occurred with the known types of problems generated by the assumption of linearity. As bugs are encountered and solved, a collection of critics is developed, each of which notices that a certain type of bug has occurred in a plan.

HACKER does a lot of wasted work. While the problem solver will eventually produce a correct plan, it does so in many cases by iterating through a cycle of building a wrong plan, then applying all known critics to suggest revisions of the plan, then building a new (still potentially wrong) plan.

---

*Comparisons with HACKER are, in a sense, unfair, since a major thrust of Sussman's work is to explore the issue of learning by experience. Many of the inadequacies of HACKER as a problem solving system arise because of its commitment to learning by subroutinization.

NOAH makes no rash assumptions, but preserves all the freedom of ordering that is implicit in the statement of a conjunctive goal. It assumes the conjuncts are independent, but the nonlinear representation frees it from worrying about additivity. It applies its critics constructively, linearizing the plan only when necessary. By waiting until it knows the nature of the conjuncts' interactions, NOAH is sure to place actions in the correct order, and thus needs never undo the effects of a false assumption.

Tate's INTERPLAN performs a search for a correct linear ordering by using both debugging and backtracking. INTERPLAN does this not by creating alternative sequences of actions, but rather by examining a tabular representation of the interactions between conjunctive goals. Tate demonstrated that a planner can perform reasoning about plans by dealing with information that is much simpler than the plan itself. This concept has been used extensively by the critics in NOAH, which do much of their analysis on the tables of multiple effects (TOMEs) rather than on the plans themselves.

Waldinger and Warren build linear plans in nonsequential order. They require that the partial plan at every stage be a linear one. However, they allow additions to the plan by insertion of new actions into the body of the plan, rather than restricting new actions to appear at the end. This approach has the advantage of being constructive, in the sense that when the planner adds each step to the plan, it takes into account all the interactions between conjuncts that it knows about. But by forcing the plan to be linear at all intermediate stages, these planners must do unnecessary search with backtracking, or sophisticated plan optimization to find the correct order in which to attack the conjuncts.

Many of the peculiarities of NOAH's problem solver arise for reasons unrelated to problem solving. The plans NOAH generates are intended for use in the cooperative achievement of tasks with a human user. This use of the plans will be discussed in the next chapter.

IV EXECUTING PLANS OF ACTION

A. Historical Perspective

The process of monitoring plan execution has received far less attention than the process of plan generation. The work that has been done (see, for example, Bolles and Paul [3]), has been mostly decoupled from planning. There have been only a few investigations of an integrated approach to plan generation and plan execution.

Nilsson [26] characterizes two broad classes of events that an execution monitor must handle: failures and surprises. Failures occur when the execution of an action fails to update the real world in the way that the model of the action updates the model of the world. Surprises occur when some fact, unrelated to the current action, becomes known. Surprises may indicate that some future steps will be unnecessary, or that they will no longer be appropriate, or that they will fail.

The most ambitious execution monitor to date has been the PLANEX system [11,12] for enabling a robot vehicle to carry out plans built by the STRIPS problem solver.

The basic interface between STRIPS and PLANEX was a tabular representation of a plan. This representation, called a triangle table, characterized for each step in the plan what its important effects were and why they were needed in the plan. Each effect (represented as a clause in the predicate calculus) was stored in a

column corresponding to the action that caused it, and in a row corresponding to the first action for which it was a precondition. The set of clauses that had to be true for any tail of the plan to be applicable, called the tail's kernel, was easy to compute.

The use of triangle tables was the first indication of the importance of the structure of a plan. By having a clear indication of which facts about the world were required for each action to be applicable, the system was able to handle failures and surprises well. Before each action was taken, the set of kernels of each remaining action in the plan was checked. The action to be executed was the first one encountered, starting from the end of the plan and working backwards, whose kernel was not satisfied. So if the world model were somehow updated to indicate that a later step in the plan were applicable, it would be immediately applied. Of course, this only helps if the world model is automatically updated to reflect any change in the real world. This was the case for the STRIPS-PLANEX system, but the overhead involved in maintaining a continuously valid symbolic model of the real world made the system run extremely slowly. A more realistic system must use its knowledge about the actions in the plan and the world it is in to know when to check the world for updates to the world model.

Nilsson [26] investigated an approach in which the distinction between plan generation and execution was deliberately blurred. His system was composed of a set of programs, called ACTIONs, that could have any or all of three kinds of effects. They could cause a real-world event to occur; they could produce or add to a plan of ACTIONs; or they could call another ACTION as a subroutine. An executive

program determined which ACTION in the current plan should be run next. In practice, the first ACTION in the plan was always run, resulting in a depth-first expansion algorithm.

The system dealt with failure by simple backtracking, and dealt with happy surprises by setting up QA4 "demons" to watch over the data base and report when an expression was asserted in the data base that achieved the goal of a subsequent step.

Nilsson's program relied on the control structure and the program structure to represent the hierarchical relationship between actions and subactions, and the time orderings between steps. The system would perform adequately on very simple problems, but it had no model of the plan it produced, and no clear idea of what it was doing. It relied on "demons" to watch for specific changes in the world model. As with PLANEX, the sytem relied on an accurate world model. For monitoring execution, one really wants demons that watch the real world and know when to update the world model.

NOAH makes a clear distinction between planning and execution but it permits a shift from one to the other at almost any time. The key to NOAH's ability to intermix planning and execution is the fact that both the input and the output of the planner and the input to the execution monitor are the same data structure: the procedural net.

NOAH deals with failures not by a simple backtracking scheme, but by a process of replanning to get the course of execution back on the track of the

original plan. The hierarchical nature of the problem solving algorithm forces NOAH to check for unpleasant surprises during planning. It should not be difficult to extend the system to make similar checks for surprises during execution as well.

B. The Basic Algorithm

We have shown in some detail how the retention of the structure of a problem enables NOAH's problem solving component to plan efficiently and without backtracking. We will now show how this structure helps in the job of monitoring the execution of plans. Then we will show how the use of a common representation for problem solving and execution monitoring enables NOAH to integrate the processes of planning and execution.

The output of the planning process is a procedural net, which was developed as a hierarchy of partially linearized plans. Figure 16a suggests the planner's viewpoint of the procedural net. A crucial factor behind NOAH's power is that the same procedural net is also the input to the execution portion of the system. The execution algorithm views the net differently, however. It sees the procedural net as a collection of action hierarchies, as suggested by Figure 16b. An action hierarchy, consisting of a node representing an action, together with its children nodes representing subactions, together with their descendant nodes, will be termed a wedge.

(a) THE PROCEDURAL NET FROM THE PLANNER'S POINT OF VIEW

(b) THE PROCEDURAL NET FROM THE EXECUTION MONITOR'S POINT OF VIEW
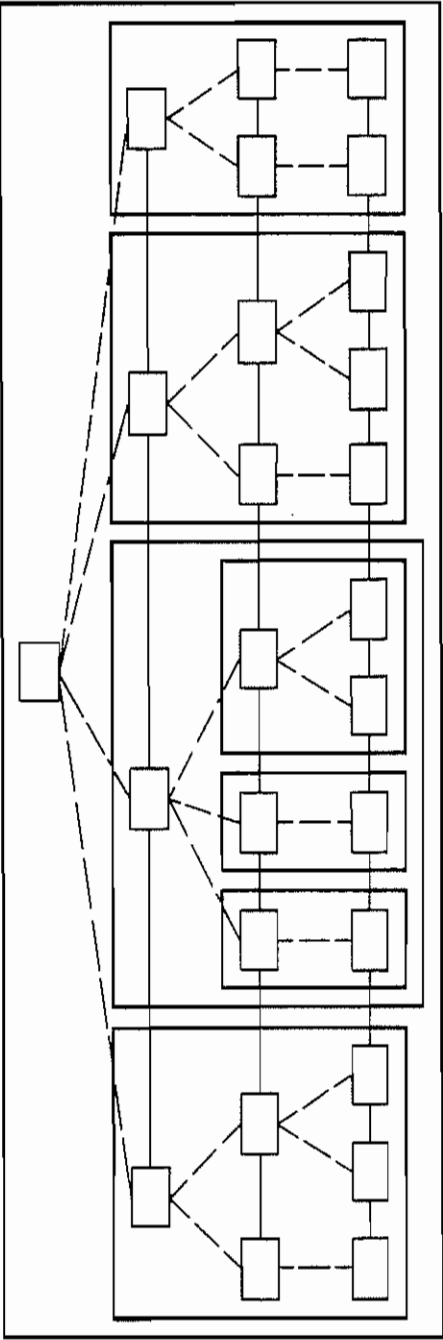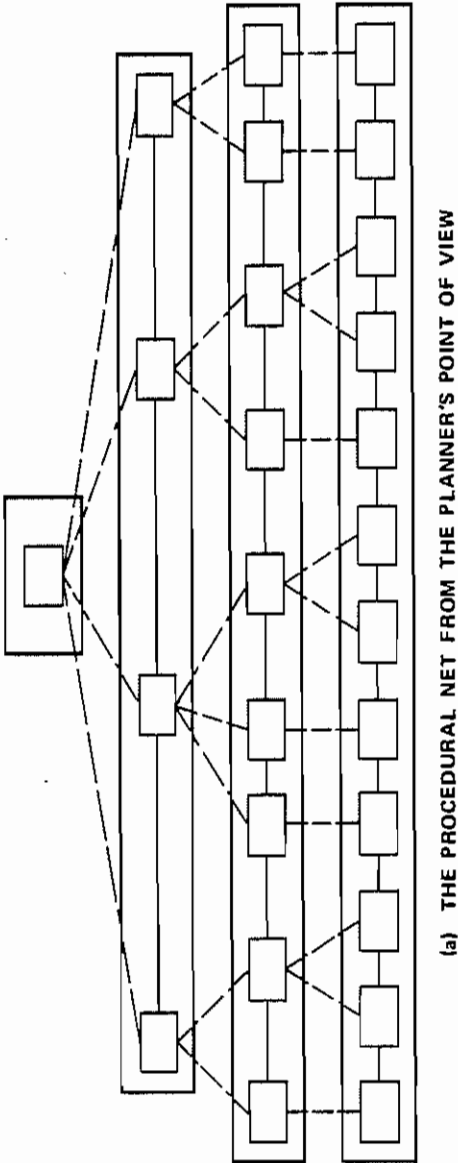
FIGURE 16   DIFFERENT PERSPECTIVES OF A PROCEDURAL NET

The execution monitor views the procedural net as a single wedge, to which it applies the following algorithm:

(1) Ask the user to accomplish the action represented by the node at the top of the wedge. This is done by saying the node's query to him.

(2) If he responds positively, assume the action has been accomplished, and so the current wedge has been successfully executed.

(3) If he responds negatively, assume he needs a more detailed breakdown of the action, and so execute in turn all the subwedges headed by children of the head of the current wedge.

Actually, the algorithm is more complicated than this. The user may make a wider variety of responses. To each query, his possible responses in the current system are:

Affirmative responses -- Yes, Ok, Yup, Check ... This type of response indicates that the user understands the instruction and is able to do it. In fact, in the current implementation, NOAH assumes that the task has been completed. This type of response signals the execution algorithm to move on to the succeeding wedge.

Negative responses -- How, No ... This type of response indicates that the user needs help before he can perform the indicated action. This signals the execution algorithm to move to the first child node.

Motivation response -- Why. The user wants to know why a certain task needs to be done. The system responds to this question by placing the task in the context of the other tasks that remain to be done at that level. This is done by

listing the queries of the succeeding sibling nodes (those that have the same parent as the current node). If the user still wants to know why, the program then repeats the query associated with the parent node. This process may be repeated (if the user asks "why" enough times) until there are no more parent nodes.

Error responses -- Help, Can't ... The user feels that the task he has just been asked to do is impossible. He feels that he must somehow be off the track of the correct plan. The system responds to this by updating its world model until the reason for the problem is discovered. Then the system plans to recover from the failure, as described in Section F below.

When the top wedge of the procedural net has been successfully executed, the execution phase terminates with "Thank You."

C. Execution Monitoring

When dealing with the real world, it is important not to assume that actions will be carried out as planned. Failures in hardware, failures in communication, and unexpected events caused by outside forces can all alter the expected outcome of an action.

The PLANEX system faced this problem by continually checking the plan's kernel against its world model, to ensure that the execution was on the right track. PLANEX presumed that an adequate mechanism existed for accurately updating the world model. This was almost the case, since there were only a small number of actions that the robot vehicle could take, and the model of each action contained information about the uncertainty it would introduce into the world model. When

uncertainties reached a threshold, the vision subsystem was used to restore the accuracy of the world model.

For the domain of the Computer-Based Consultant, or even for a richer robot domain, this approach will prove inadequate. The number of ways in which an action may go wrong is very large. The apprentice or a semi-autonomous robot may initiate actions independently. In the CBC system, the apprentice is relied on to provide much of the information about the real world, and he may provide misinformation. So NOAH cannot treat the world model as a given. It must initiate interactions with the user at appropriate points to ensure that it is accurately monitoring the course of the execution.

To do this dynamically during the course of execution requires a model of the user's capabilities. This is not presently implemented in the NOAH system, so this type of execution monitoring is not done. However, when a serious error is discovered (requiring the system to be more thorough in its efforts to determine the state of the world), the system must determine what portions of its world model differ from the actual situation.

The system does this by asking the user to verify that he has carried out the portion of the plan that precedes the action where the error was discovered. The verification is done hierarchically, so that the system verifies the higher level plans before verifying the plan at the level where the error was discovered. Each plan is verified by querying the user about all the actions that precede the current action. The requests for verification are of the form: "Did you ...", followed by the query of the node that represents each action. The verifications are carried out in reverse

chronological order. To each request for verification, the user may give one of three classes of response:

Positive responses -- Yes, Ok, Yup, Check ... In this case the system has learned nothing useful, since the user thought he had done the action correctly in the first place, and yet some action must have been performed incorrectly. The verification algorithm goes on to other nodes.

Negative responses -- No, ... In this case the discrepancy between the system's model and the real world has been found.

Unsure responses -- Can't, Help ... The user may indicate that he is unsure, in which case the system immediately provides more detailed questions by asking him about the node's children.

If all the preceding actions at the level where the error was noticed have been verified, and all the higher level plans have been verified, and the user still feels nothing is wrong, the system will ask him to verify all the more detailed plans, in order of increasing detail. If all the plans in the net have been verified, and the error still has not been found, the system will extend the procedural net by expanding the partially executed plan as described in Section D below. If no extension is possible because the action semantics contain no more detail, the system gives up.

This algorithm tries to home in on the particular problem using minimal interaction with the user. It avoids going into details unless no information has been found at a general level, or unless the user requests them.

D. Expanding a Partially Executed Plan

The simple problem solving algorithm described in Chapter II expanded a procedural net completely, until no new details were uncovered. This is often impractical, especially when the plans are used to interact with a partially skilled human who can carry out some detailed planning himself. An ultimate version of NOAH would make a local decision at every node about whether to expand it. The current version allows the expansion to terminate when a predetermined depth is reached.

Now, the human user may run into trouble at any point, and require instructions that are more detailed than have been planned for. But as long as there is a body of code associated with the most detailed nodes, the system can simply continue building the plan. Rather than insisting on building an entire level at a time, the system can simply expand the relevant actions. This is done by expanding the node without criticism, a relatively cheap operation.

This operation is accomplished by first building a dummy PLANHEAD node that contains the image of the world as seen by the node being expanded. The new subplan is built following this dummy node. The existence of the dummy node allows the new subplan to be developed in a world model that corresponds to that of the node being expanded.

In the transcriptions of plan execution to be presented below, the expansion of a node whose query is query is indicated as:

Expanding node-number: query .

The capability to expand a plan at execution time is especially important for applications involving interactions with humans. The system must have the potential for handling a large number of details, without actually doing all the work to plan for them. Section G below will show examples of this feature.

E. Executing Iterative Actions

The general plan expansion algorithm described in the preceding sections uses the semantics ebodied in the SOUP code to generate detailed knowledge about actions. For dealing with the non-initial passes around a loop, a much simpler, syntactic approach is adequate.

As described in the chapter on planning, an iterative action is expanded by NOAH into two components: a detailed subplan for a single pass around the loop and a special REPLICATE node that contains the information necessary to compute subplans for subsequent passes. The query of the REPLICATE node is identical to that of the node specifying the original iteration, except that any references to n, the number of times the loop is to be traversed, are replaced by the number n-1. (If n equals 1, no REPLICATE node is included, since there is no need to continue the iteration.)

The algorithm for executing a subplan that begins with an expansion of an n-way iteration is as follows:

(1) Set the variable i to 1.

(2) Execute the subplan for the ith pass through the loop.

(3) Ask the apprentice to execute the remaining iterations by presenting the query of the REPLICATE node.

(4) If the apprentice responds positively, we are done.

(5) Otherwise, use the exemplar and the formal objects list of the REPLICATE node to generate a subplan for the i+1th pass through the loop. Update the REPLICATE node to specify n-i remaining iterations. Go to step 2.

As an example, Figure 17 presents three dialogues that might be generated from the plan that was presented in Section E of Chapter 3, for installing the belt housing cover of an air compressor.

FIGURE 17

THREE DIALOGUES GENERATED BY THE PLAN OF FIGURE 15

Dialogue 1

```
<(SCREW BELTHOUSINGCOVER TO BELTHOUSINGFRAME)
>HOW
<(GET 10 1/2 INCH #8 PAN-HEAD SCREWS)
>OK
<(LOOSELY FASTEN THE 10 SCREWS)
>OK
<(TIGHTEN THE 10 SCREWS)
>OK
```

FIGURE 17

THREE DIALOGUES GENERATED BY THE PLAN IN FIGURE 15 (CONCLUDED)

Dialogue 2

```
<(SCREW BELTHOUSINGCOVER TO BELTHOUSINGFRAME)
>HOW
<(GET 10 1/2 INCH #8 PAN-HEAD SCREWS)
>HOW
<(GET ONE 1/2 INCH #8 PAN-HEAD SCREW)
>HOW
Expanding NODE177: (GET ONE 1/2 INCH #8 PAN-HEAD SCREW)
<(LOOK IN THE SMALL CABINET IN THE DRAWER MARKED #8)
>OK
<(GET 9 1/2 INCH #8 PAN-HEAD SCREWS)
>OK
<(LOOSELY FASTEN THE 10 SCREWS)
>OK
<(TIGHTEN THE 10 SCREWS)
>OK
```

Dialogue 3

```
<(SCREW BELTHOUSINGCOVER TO BELTHOUSINGFRAME)
>HOW
<(GET 10 1/2 INCH #8 PAN-HEAD SCREWS)
>OK
<(LOOSELY FASTEN THE 10 SCREWS)
>OK
<(TIGHTEN THE 10 SCREWS)
>HOW
<(TIGHTEN ONE SCREW)
>HOW
Expanding NODE185: (TIGHTEN ONE SCREW)
<(GET THE SCREWDRIVER WITH 1/4 ON THE HANDLE)
>OK
<(TURN THE SCREW CLOCKWISE WITH THE SCREWDRIVER UNTIL FAIRLY TIGHT)
>OK
<(TIGHTEN THE 9 SCREWS)
>OK
```

F. Error Recovery

A major concern of all systems that must deal intelligently with the real world is the ability to recover gracefully from unexpected events in the real world. The representation of actions in a procedural net allows NOAH to home in on the nature of the unexpected event, as described in the preceding section, and to plan for new actions to allow as much as possible of the existing plan to be reused.

The current implementation has a rather simple-minded approach to error recovery. It is intended to be an indication of the adequacy of the procedural net to support this aspect of intelligent behavior, rather than a demonstration of good ideas about error recovery.

When an error is pinpointed, as described in the previous section, NOAH creates a new plan to achieve the intended effects of the erroneous action. This is done in a manner similar to that done when expanding an existing plan to greater detail. There are a few important differences, however. First, the dummy PLANHEAD node contains an image, not of the current action's branch in the partial ordering, but of the world model built up from all the actions that were executed. Secondly, since the new nodes are at the same level as existing nodes, rather than being more detailed as in the simple extension case, the new subplan must be integrated into the existing plan. This may involve recursive calls on the replanning algorithm since redoing the effects of the erroneous action may undo desired effects of other actions.

G. An Example

Rather than give detailed algorithms for error recovery, we shall simply present an example dialogue that the system generated when recovering from an error. The task was to assemble an air compressor from a partially disassembled state, and the first three levels of the procedural net are shown in Figure 18. The error on the user's part was in installing the pump backwards. While it is unlikely that a real user would make such a mistake, similar problems are extremely likely when dealing with subcomponents of a device like the air compressor. In fact, the example was inspired by a real problem taken from a protocol of a partially skilled apprentice installing the pump, in which the top of the pump had been installed backwards, allowing the pump to be fastened to the compressor without trouble, but rendering later steps in the installation impossible. We could have used this example, but semantics have not yet been developed for the components of the pump.

```
>(TRY TO ACHIEVE ASSEMBLED AIR COMPRESSOR)
 <HOW
(We will INSTALL PUMP and INSTALL PUMPBRACE and INSTALL PUMPPULLEY
   and INSTALL BELT and INSTALL BELTHOUSINGCOVER)
        The various INSTALLs have been so intermixed by criticism
        that there is no meaningful order in which to do them.
        So the system mentions that they will all be done, and then
        drops immediately to a more detailed level to specify
        individual actions.
>(CONNECT BELT TO MOTORPULLEY)
 <OK
>(CONNECT PUMP TO PLATFORM)
 <OK
        The apprentice thinks he has performed this step correctly.
        Actually, he has installed the pump backwards.
>(CONNECT PUMPPULLEY TO PUMP)
 <OK
        This can be done even though the pump is on backwards.
>(CONNECT BELT TO PUMPPULLEY)
 <CAN'T
        This step cannot be done; the system then tries to correct
```

100

LEVEL 1

```
┌─────────────────────────┐
│      Try To Achieve     │
│ (Assembled Aircompressor)│
└─────────────────────────┘
```

LEVEL 2

LEVEL 3
(Before Criticism)
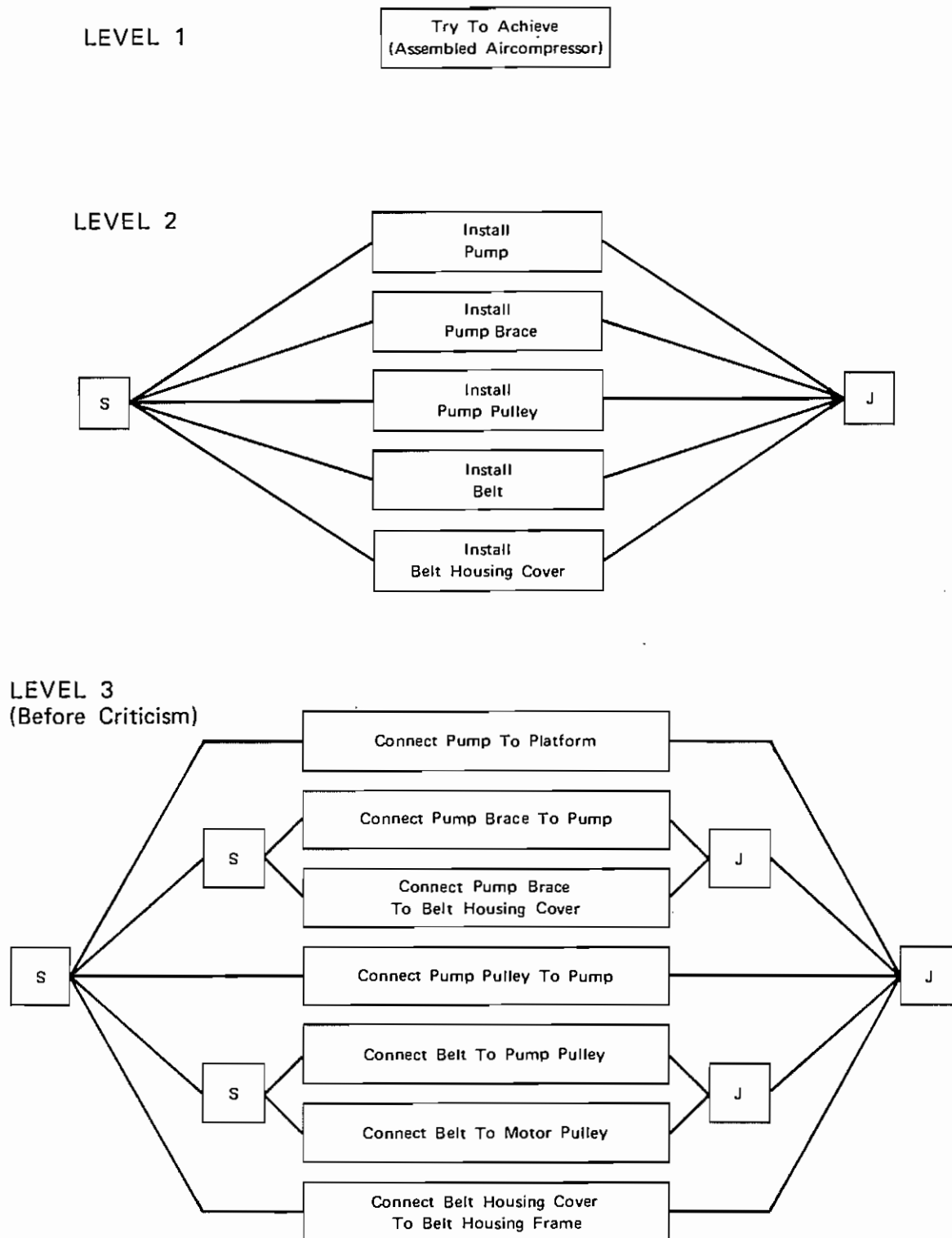
FIGURE 18   PROCEDURAL NET FOR AIR COMPRESSOR ASSEMBLY

LEVEL 3
(After Criticism)
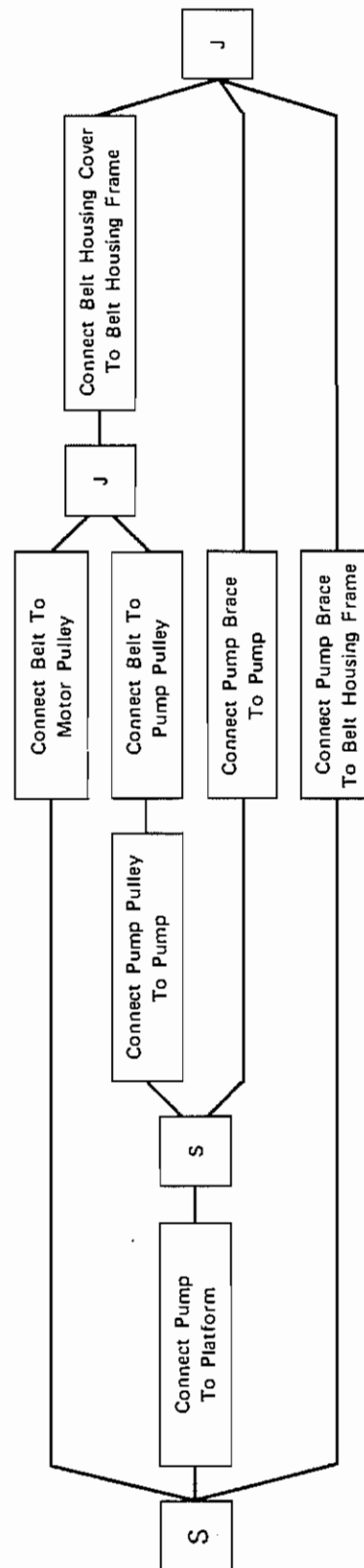
FIGURE 18   PROCEDURAL NET FOR AIR COMPRESSOR ASSEMBLY   (Concluded)

its model of the world by querying the apprentice about the
expected state of the world at this point in the execution
of the plan. Using the algorithm specified in Section C,
the system asks questions at levels no more detailed than
that of the step that couldn't be executed. It uses the
portions of the procedural net shown in Figure 19.
>(Did you INSTALL PUMPPULLEY)
 <YES
>(Did you CONNECT PUMPPULLEY TO PUMP)
 <YES
>(Did you INSTALL PUMP)
 <HELP
        The apprentice indicates that he is unsure if he installed the
        pump correctly. This aids in directing the system toward the
        problem. The system would soon arrive at the problem even if the
        apprentice said "YES".
>(Did you CONNECT PUMP TO PLATFORM)
 <HELP
        The system now utilizes information in the procedural net about
        more detailed actions, as shown in Figure 20.
>(Did you ATTACH PUMP TO PLATFORM)
 <YES
>(Did you POSITION PUMP WITH RESPECT TO PLATFORM)
 <HELP
Expanding NODE44: (POSITION PUMP WITH RESPECT TO PLATFORM)
        The level of detail required now is greater than the level to
        which the procedural net has been built. The system
        dynamically expands the net, using the algorithm described in
        Section C, to provide greater detail about the action in
        question. The procedural net now appears as in Figure 21.
>(Did you PLACE THE PUMP OVER THE BOLTHOLES IN PLATFORM SO THAT THE
 OILDRAINNIPPLE IS ORIENTED FORWARD)
 <NO
        The error has been discovered. Now the system will plan to
        recover from it.
(Expanding plan at level 3)
Expanding NODE73: PLANHEAD
Expanding NODE74: (Reachieve CONNECTED PUMP PLATFORM)
(Expanding plan at level 4)
Expanding NODE75: PLANHEAD
Expanding NODE76: (DISCONNECT PUMPPULLEY FROM PUMP)
Expanding NODE77: (DETACH PUMP FROM PLATFORM)
Expanding NODE78: (POSITION PUMP ON PLATFORM)
Expanding NODE79: (PLACE THE PUMP OVER THE BOLTHOLES IN PLATFORM SO
        THAT THE OILDRAINNIPPLE IS ORIENTED FORWARD)
        The new plan is shown in Figure 22.
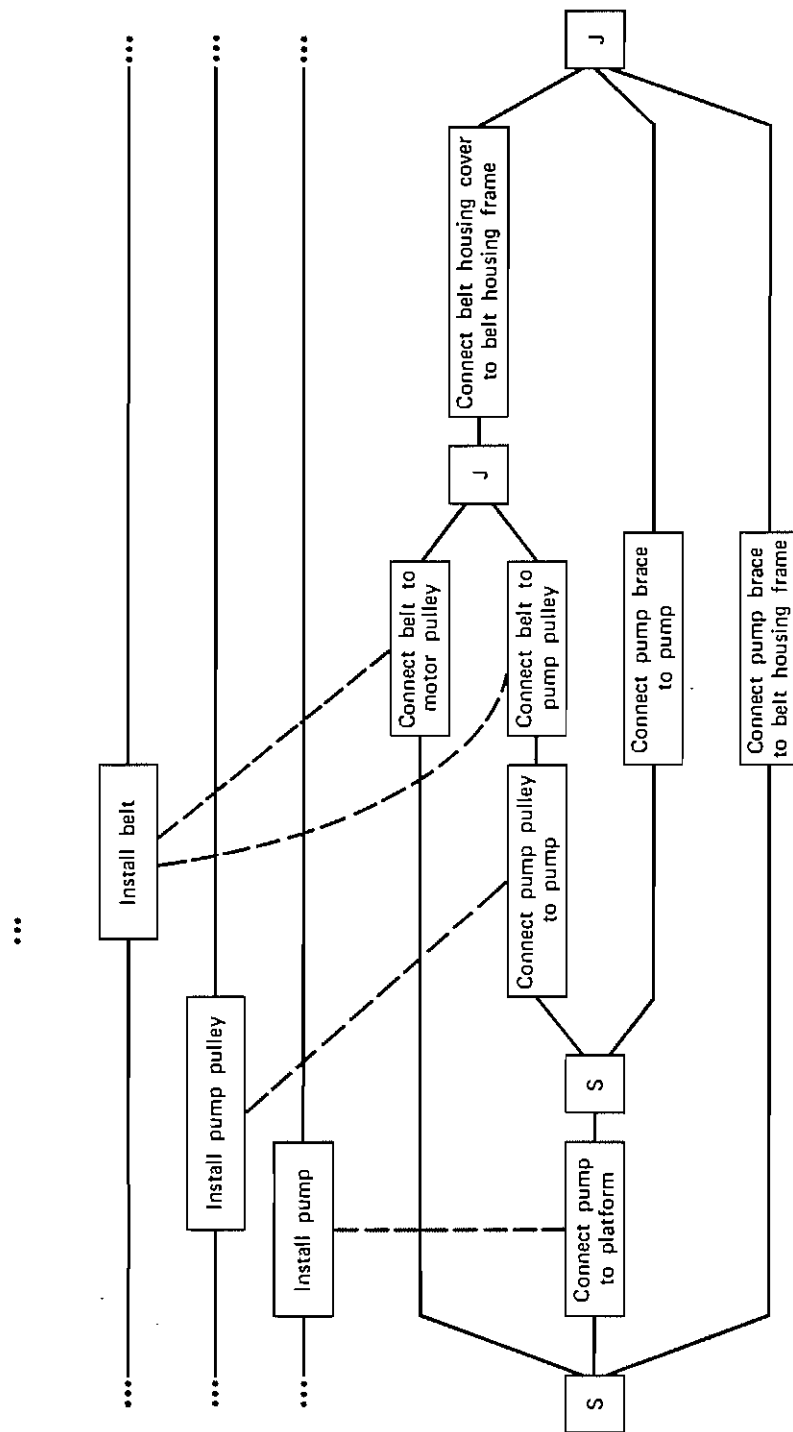
FIGURE 19 PORTION OF PROCEDURAL NET USED FOR PRELIMINARY QUESTIONS

104
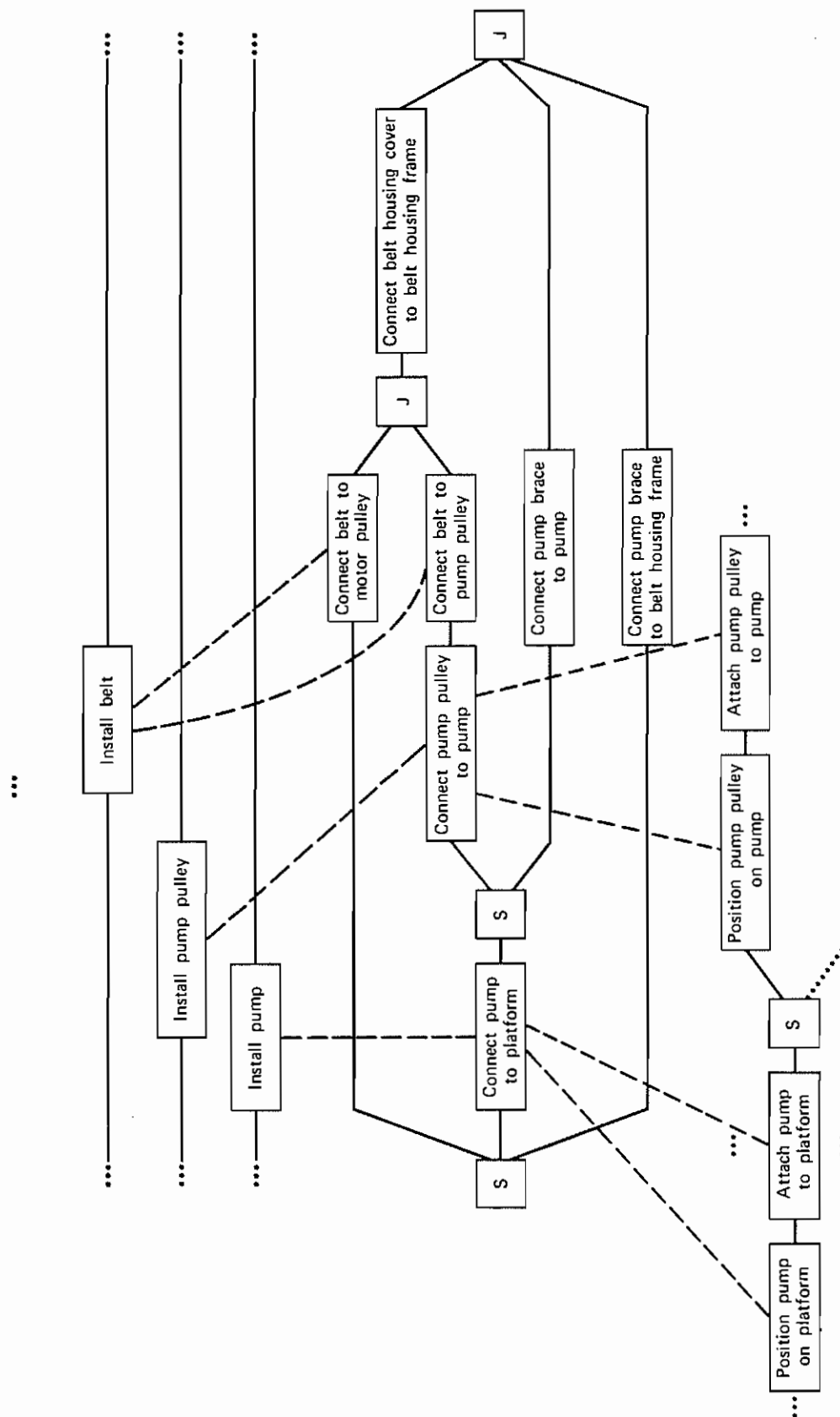
FIGURE 20   PORTION OF PROCEDURAL NET USED FOR SUBSEQUENT QUESTIONS
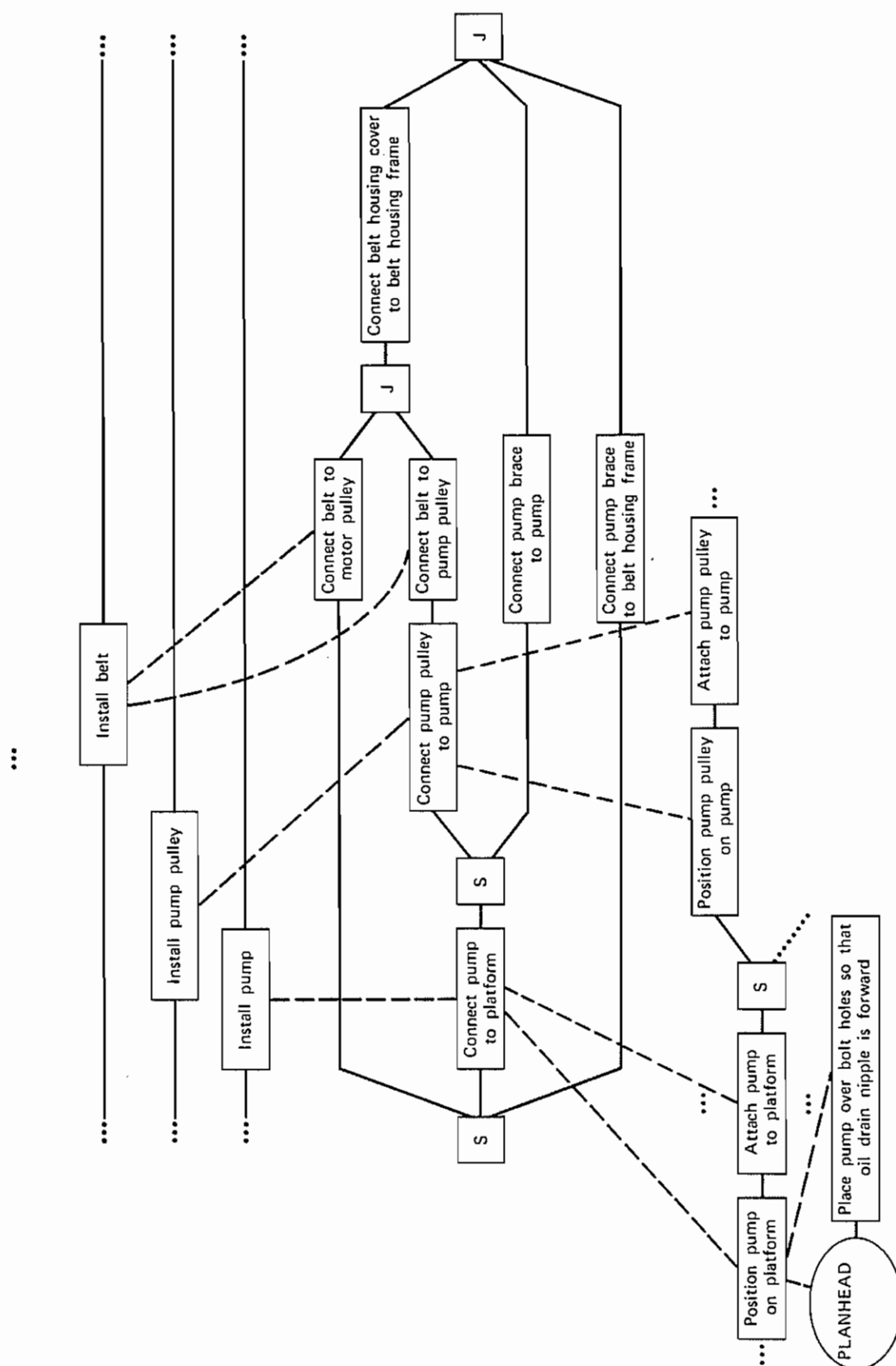
FIGURE 21   PORTION OF PROCEDURAL NET INCLUDING NEWLY GENERATED SUBPLAN
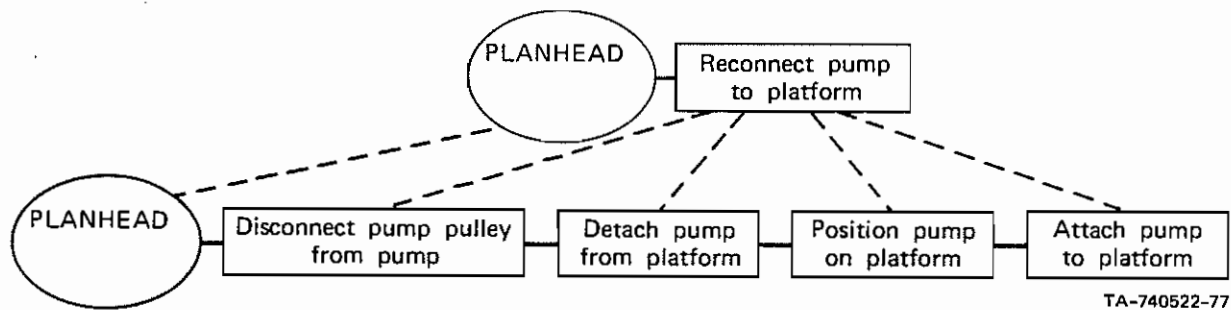
FIGURE 22   PROCEDURAL NET TO RECOVER FROM ERROR

Patching in NODE74
> The new plan is now patched into the existing plan, as
> suggested in Figure 23. The system discovers that the new
> plan would disconnect the pump pulley from the pump in the
> course of positioning the pump correctly. So it creates a
> new plan for attaching the pulley again.

(Expanding plan at level 5)
Expanding NODE86: PLANHEAD
Expanding NODE87: (Reachieve CONNECTED PUMPPULLEY PUMP)
> The new plan is patched in.

Patching in NODE87
> The patched plan, shown in Figure 24, will now fix the error
> and its side effects.

>(Reachieve CONNECTED PUMP PLATFORM)
 <HOW
>(DISCONNECT PUMPPULLEY FROM PUMP)
 <OK
>(DETACH PUMP FROM PLATFORM)
 <OK
>(POSITION PUMP ON PLATFORM)
 <HOW
>(PLACE THE PUMP OVER THE BOLTHOLES IN PLATFORM SO THAT THE
  OILDRAINNIPPLE IS ORIENTED FORWARD)
 <OK
>(ATTACH PUMP TO PLATFORM)
 <OK
>(Reachieve CONNECTED PUMPPULLEY PUMP)
 <OK
> The patch has been executed, so execution proceeds to
> conclusion.

>(CONNECT BELT TO PUMPPULLEY)
 <OK
>(CONNECT BELTHOUSINGCOVER TO BELTHOUSINGFRAME)
 <OK
>(CONNECT PUMPBRACE TO PUMP)
 <OK
>(CONNECT PUMPBRACE TO BELTHOUSINGFRAME)
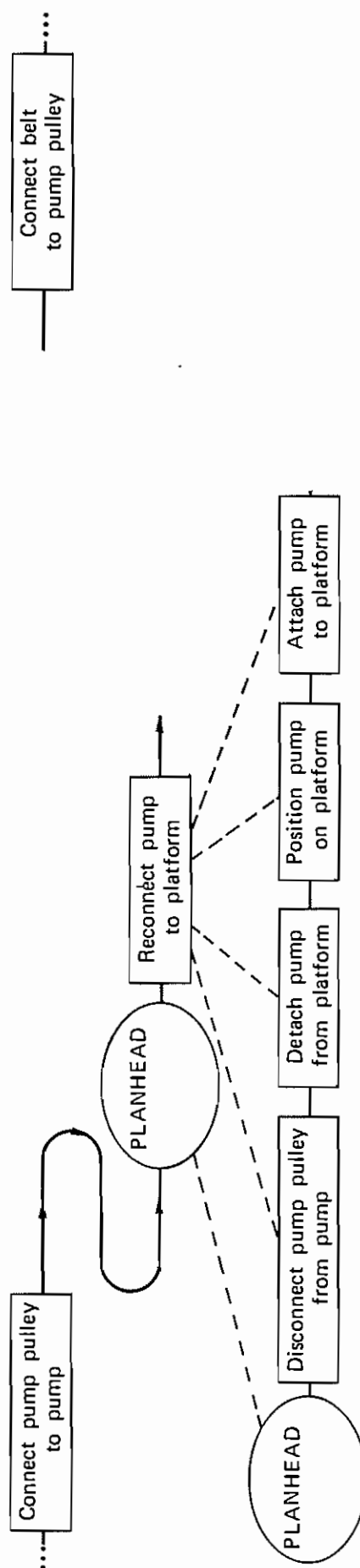
FIGURE 23   PROCEDURAL NET WITH NEW PLAN PARTIALLY PATCHED IN

108
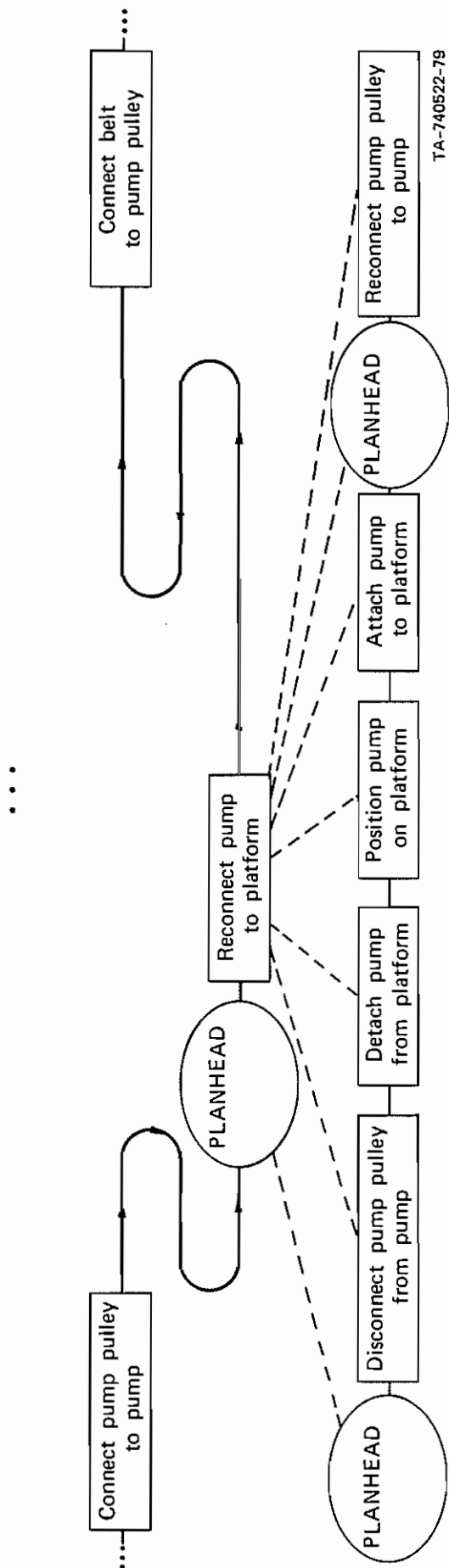
FIGURE 24   PROCEDURAL NET WITH COMPLETED PLANS FOR ERROR RECOVERY

TA-740522-79

<OK
(Okay, we did INSTALL PUMP and INSTALL PUMPBRACE and INSTALL
  PUMPPULLEY and INSTALL BELT and INSTALL BELTHOUSINGCOVER)
Thank you

In this example, we have seen NOAH provide plans at varying levels of detail
to match the apprentice's ability. We have seen the system update its model of the
world when a serious failure took place during execution. And we have seen NOAH
recover from the failure by creating and then executing a new hierarchy of plans to
get back on the path of the original plan.

H. Comparison with Other Current Work

NOAH relies heavily on the rich structure of the procedural net to enable it to
monitor execution and to recover from failures. A somewhat similar representation
was used by Hayes [17] for a system that created and simulated travel plans.
Hayes' representation involves two related data structures: a tree of subgoals and a
graph of "decisions". The subgoals, for instance "go from Paris to Marseilles by
train," are represented in a tree structure identical to that created by the parent-
child links of the procedural net. The decisions, for instance, "travel from Paris to
Nice via Marseilles (as opposed to some other city)," are stored in a graph structure
where links represent logical dependency. Associated with the decisions are
pointers to the subgoals that are their direct consequences.

When a failure of the plan occurs, Hayes' system replans by removing the
portion of the plan that has been executed, identifying the decisions that are
inappropriate in the new situation, eliminating their effects from the plan, and then
using whatever is left of the original plan as a basis for constructing a new plan.

This is a very conservative approach, as it will throw out any portion of the plan that might possibly be affected by an unexpected real-world event. For rather complex plans this is likely to be unacceptably inefficient. On the other hand, the explicit representation of the decisions the system made is extremely useful. Even if a more sophisticated recovery mechanism were to be more deliberate about what portion of the plan was to be thrown out, the graph of decisions provides exactly the right information to do it.

It is worth noting that, although the strategies employed by NOAH for execution monitoring and error recovery are rather simple, the behavior generated by the system is rather impressive to a naive viewer.

V INADEQUACIES OF THE CURRENT IMPLEMENTATION

The NOAH system is a testbed for exploring a simple concept: that the structure of a plan can be much richer than existing systems have presumed, and that this structure can be exploited both in solving problems and in monitoring the execution of their solutions. The system described here is primitive and incomplete, and a more complete one will be required to fully explore the implications of this representation of plans. This chapter will discuss some of the more serious inadequacies of the current implementation.

A. The Generality of the Procedural Description of Domain Knowledge

The most serious deficiency in the current system is its lack of awareness about the auxiliary computations specified in the procedural semantics (the SOUP code) of a task domain. The procedural net representation lets the system be aware of the goals and subgoals that the planner has decided to tackle, but it does not preserve any information about the computation that resulted in those decisions. In some cases, a reordering of subgoals might alter the state in which one of these computations would be carried out. Then the computation might produce different results.

For example, a section of SOUP code for depositing a check at the bank might say, "If you are driving a car, then go to the drive-up window. Otherwise, go inside. Then deposit the check." Now, a parallel subplan might require the use of a car, or might require that some other business be transacted on foot right next to the bank.

The choice of whether to go inside the bank should not be based on whether the world model showed that a car was in use at the time the "Deposit the check" action was expanded. The choice of the "Go to window" subgoal or the "Go into bank" subgoal should be affected by the operations in parallel branches of the plan. But unless the system is explicitly aware of the nature of the conditional test, it will already have irrevocably chosen a particular subgoal by the time the critics have a chance to consider all the parallel branches together.

There are two ways in which the deficiency could be dealt with. One approach would be to restrict the complexity of the SOUP code that specifies the actions of a task domain. Then the system would not alter the effects of an action merely by changing its position in the plan. For example, the conditional test might be forbidden, so that, for example, the drive-up window would always have to be used.

However, if NOAH is to be effective in truly complex domains, SOUP must have all the richness of a PLANNER-like language [2], and the system must be aware of this new type of interaction. So the ultimate NOAH must take an alternative approach, and provide a mechanism for noting these auxiliary computations. A notation of all decisions taken during planning, such as Hayes [17] used, would provide the necessary information. But the number of decisions made in the course of evaluating PLANNER-like functions is relatively large compared to the small number of decisions that the system would actually need to notice.

Perhaps a more workable solution would be to allow the entries in the tables of multiple effects to specify a computation as well as a simple expression. The computation, evaluated at the time a critic is analyzing interactions, would reflect the effects of the currently postulated order of subgoals.

Analysis of the many alternative tables that this approach would generate might be too expensive computationally. Some sort of network analysis might prove in the end to be more satisfactory than the tabular approach that is used in the current system.

B. Relation of Hierarchical Model and Action Hierarchy

The progressive expansion of plans puts a set of constraints on the interrelationship of the declarative model and the procedural descriptions of actions. The critics will not notice an interaction between actions unless the same predicate name is mentioned by them. So it is of some importance that functions that are nested equally deeply in the procedural semantics refer to the same set of predicates. If not, the interaction between the actions will not be noticed by the system until the deeper of the two actions refers to the set of predicates. If the plan should not be carried down to a sufficiently deep level of detail, the system may never notice the interaction at all and produce an incorrect plan.

The solution to this problem would require the system to treat the effects of an action on a semantic basis rather than the syntactic one used now. Rather than noticing interactions just on the basis of a (syntactic) pattern match, the system would have to use a mechanism such as that proposed by Fikes [10] to compute whether any effect caused by an action in a plan might imply any other effect in the plan, or the negation of such an effect. Alternatively, it might use a regression technique such as Waldinger [39] uses, with very detailed regression rules to reflect all possible side-effects of each action. This clearly would be unacceptably expensive unless it were carefully controlled by some other mechanism.

The basic point of this discussion is that the system derives a large measure of its power from the simplicity of its algorithms for noticing and dealing with multiple effects. When these algorithms are forced to move from a strictly syntactic approach to one that permits even shallow deduction about the effects, the power of the system may be reduced.

C. Backtracking and Other Forms of Search

NOAH currently makes no provision for backtracking if a detailed expansion of a plan indicates that the higher level plan won't succeed. (This is not quite true; the system can try alternatives specified in a SOME node.) The research reported here has focused on the non-search aspects of problem solving, but the procedural net representation can certainly support a provision for backing up when a failure occurs. This will have to be done if NOAH is to be truly competent at solving problems. It is interesting to note, however, that with an appropriate ranking of actions into a hierarchy, domains as complex as the CBC can be encoded in a way that requires search only within a given level of the plan.

VI EXTENSIONS TO THE CURRENT IMPLEMENTATION

The richness of the structure of the procedural net representation allows knowledge about actions and plans of actions to be stored in a computer memory and used in a variety of worthwhile ways. The current NOAH system stores a few kinds of knowledge and exploits them in a few of these ways. This chapter will begin by discussing several other kinds of knowledge that are important for problem solving and execution monitoring, and that can be used with particular effectiveness when stored within the structure of the procedural net. Next will follow a discussion of uses of the structure for natural language understanding and for learning. Finally, a few other extensions of the present work, relating to other aspects of the current system will be presented.

Some of these extensions are relatively easy to implement; others are major research projects in their own right. The purpose of this chapter is to raise questions, not to suggest answers.

A. Nodes as Action Frames

As we mentioned in the section on representation, the nodes of the procedural net can be viewed as instantiations of frames about actions. They are a limited kind of frame, however. All the "slots" for associating entities with a node in the net are derivable from either the SOUP code associated with the action or from the structure of the net itself. There are many other useful kinds of information that could be associated with individual nodes. None have thus far been included in the

NOAH system because no mechanism now exists for specifying their association with individual nodes. Thus, the inclusion of the features discussed in this section would reduce the simplicity of the current procedural method of specifying the semantics of a domain.

Among the items that might be associated with a node is a specification of postconditions. Postconditions are the analogue of preconditions; they are a set of items that must be established upon completion of an action. An example of a postcondition is that a pulley should rotate without wobble after being installed. No aspect of the installation process need refer to the potential of a wobble; indeed, there is no prevent-wobble action. But a check about wobbles is appropriate at the end of the installation. The use of postconditions can be very important in execution monitoring, and might also aid in enriching the world model for use in planning and error recovery.

This particular item is rather easily represented in the existing framework for specifying semantics. The current CBC system associates preconditions with particular goal expressions. The same approach could be used to associate postconditions with particular goals.

Another item of particular importance in applications with human interaction is a set of expected ways in which the action could be erroneously executed. Minsky [25] has discussed the importance of such a feature for his frame systems. This would help the system to catch errors at the time they occur, thus avoiding many instances where extensive replanning for error recovery is necessary. The individual possibilities of failure could even be matched against a model of the ways

in which the user has erred in the past to determine if there are any specific additional warnings that should be given during execution. Since the set of these possibilities is likely to be quite large, it would not be practical to encode them directly into the SOUP code. Some alternative method would be preferred for associating them with particular nodes in the net.

An item of great importance for a system that could interact verbally with a human user is information about expected queries that a human user might ask during the execution of a particular action. By associating with an action a small set of related concepts, or even key words to search for in the speech stream, a great heuristic advantage could be provided to a speech understander. Again, the mechanism for associating this list with a particular action is not obvious.

Another type of item that might play a strong roll in diagnosis-oriented applications is a reference to an alternative representation of the state of the world. The basic planner and execution monitor relies on a representation of the world model as a collection of expressions. But other ways of representing aspects of the world, for instance a characterization of an expected view of a scene when an action is completed, a function specifying the gradual effects of an action over time, or the acoustic pattern produced by running machinery, might be associated with some nodes in the net. The alternative representations would augment the system's ability to understand and reason about actions.

B. A Task Model for Natural Language Understanding

Discourse, especially spoken discourse, is replete with partial utterances and

sentences that are ambiguous when taken in isolation. Recent approaches to understanding natural language (see, for example, Chafe [6]) have stressed the importance of using a history of the dialogue and a current world model to aid in completing partial utterances and in resolving ambiguous references. For task-oriented dialogues such as those involved in the cooperative efforts of an apprentice and the Computer-Based Consultant, it has been shown by Deutsch [7] that a richly structured representation of the task at hand is an important additional source of knowledge.

For example, consider the following fragment of a dialogue between a human expert and a human apprentice, taken from Deutsch:

> Apprentice: One bolt is stuck. I'm trying to use both the
> pliers and the wrench to get it unstuck, but I
> haven't had much luck.
> Expert: Don't use pliers. Show me what you are doing.
> Apprentice: I'm pointing at the bolts.
> Expert: Show me the 1/2" combination wrench, please.
> Apprentice: OK
> Expert: Good, now show me the 1/2" box wrench.
> Apprentice: I already got it loosened.

The apprentice was not reporting that he had loosened the box wrench or the pliers. It is obvious that he's loosened the bolt. The pronoun reference would be hard to resolve by considering the dialogue in the linear sequence in which it is produced. But when the structure of the task being carried out is considered along with the dialogue, resolving the reference is much easier.

Figure 25 displays a procedural net that might represent the example subtask at the time the apprentice said, "I already got it loosened." The utterances of the expert and the apprentice are associated with the appropriate nodes. The lower
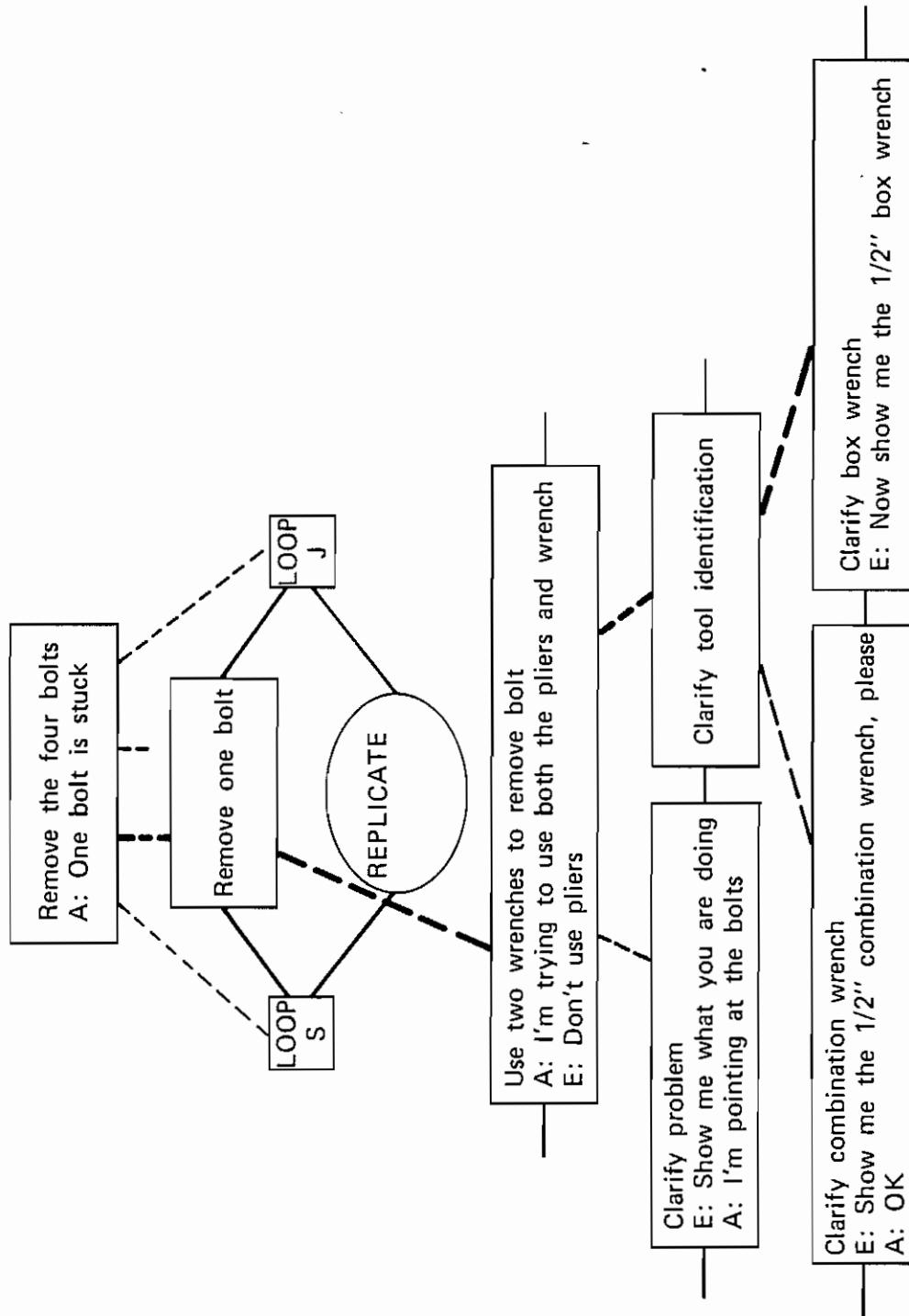
FIGURE 25   PROCEDURAL NET WITH UTTERANCES ASSIGNED TO NODES

four nodes would have been generated during the course of execution; the higher nodes are part of the standard plan for the subtask. Deutsch claims that the only utterances that need be considered in resolving a pronoun reference are the last utterances at each level of the detail hierarchy. The sequence of nodes that must be considered for the resolution are indicated by bold parent-child links in the figure. By analyzing the small number of relevant utterances, one can easily understand that the reference is to the bolt. The procedural net has been designed with its potential use as a task model in mind. During execution, a set of links is established that trace the course of the actual execution through the plan. Thus, at any point during the course of execution, the system can have access to the specific execution path taken by the apprentice through the plan. This real-event linkage, in conjunction with the predecessor-successor and parent-child links of the original hierarchical plans, constitutes a rich model of the task and its course of execution. When coupled with a history of the dialogue, which may be distributed among the nodes as shown in our example, the structure of the task will become an important aid in understanding discourse.

The speech understanding component of the Computer-Based Consultant will utilize this task model as an aid in resolving ambiguities and completing partial utterances. In addition, the task model will be of great value in the inverse problem of generating descriptions of objects. The computer based consultant must speak unambiguously, yet parsimoniously. Deutsch's work suggests that to avoid ambiguity without being overly descriptive, the consultant need only distinguish among objects that are referred to in that subset of the currently active wedge that represents actions at varying levels of detail that are currently under completion.

C. Saving and Reusing Plans

There have been surprisingly few serious attempts at machine learning of programs or plans of action. Many systems that can build plans or programs might well be extended to save and attempt to reuse the results of their efforts. Unfortunately, learning is a hard problem, and most workers (the author included) have chosen to tackle other problems.

The two major successes in learning about sequences of actions in the last five years have been the STRIPS system of Fikes, Hart, and Nilsson [12] and Sussman's HACKER [35]. Both of these systems learned by creating entities with much more structure than the original plan itself. For STRIPS, a MACROP was created that represented a sequence of actions in aggregation. Although the original plans were linear sequences of actions, the MACROPS were represented as "triangle tables" that represented the interrelation of the preconditions and effects of each action with all the others, not just that of the actions that were sequential in the original plan. For HACKER, a subroutine was created or modified that had, in general, a more complex structure than the particular plan the subroutine would generate given the current problem.

While this section will only make vague proposals about learning and reusing plans, it is worth noting that the procedural net already represents a plan in a very rich and well-structured way. Since the structure is rich enough to enable the NOAH system to perform both routine execution monitoring and more sophisticated error recovery, there is some evidence that it is rich enough to adequately represent a learned plan (or hierarchy of plans) of action. Let us explore the

characteristics that determine when a plan represented as a procedural net is worth learning. The expansion phase of the planning process is extremely efficient; there is little point in simply storing the results of expansion. The criticism phase, when a global look is taken at the overall plan, is more expensive, however. Thus, a plan that had been subjected to severe criticism would be a good candidate to save for possible future reuse.

NOAH is very conservative about altering plans during criticism. Only reorderings that are forced by conflicts in the plan structure, or mergings that are justifiable within the scope of the criticism are performed. Any conflict in a criticized subplan would also be a conflict in the new more global plan. Any merging that was valid in the subplan would be valid in the larger plan. The ideal plan to save for possible future use, then, is one in which a substantial amount of criticism has taken place at several levels of detail.

How can the system characterize the newly saved wedge to determine when it is relevant to insert in a new plan? The saved plan was completed in the exact way it was because it was computed in the context of a particular world model. Those aspects of the world model that were used during the computation of the plan at each level of detail can be aggregated on the add and delete lists of a dummy PLANHEAD node at the beginning of each plan. If the effects of the entire wedge (characterized by the add and delete lists of the node at the top) are desired, and the current world model at a number of levels matches the one under which the wedge was originally computed, then the wedge is applicable and can be patched into the existing plan just as error recovery plans were patched in. The presumed

world model at the beginning of the plan at each level even tells the system how many levels of the wedge can be validly patched in, and indicates where fresh computations in a new, detailed environment must be made.

The hierarchical structure of the saved wedge would hopefully allow the system to use the learned sequence of actions when it was approximately relevant as well as when it was exactly relevant.

D. Learning New Semantics Through Interaction with an Expert

Yet another type of learning is also important, especially for expert systems such as the Computer Based Consultant. Expert systems cannot be encoded in one shot. They require a good deal of interaction with one or more experts to debug and expand their expertise until they begin to perform competently. So an expert system needs a way to acquire and easily encode new expertise, and to debug old attempts at expertise. Existing expert systems that accept updating readily, such as DENDRAL [4] and MYCIN [33], have very small, localized primitive chunks of knowledge that all adhere to a rigid syntax. This makes it easy to replace or update an entire chunk. It also reduces the heuristic power of the systems, because the grain of each individual action is so fine. The NOAH system has a much more powerful primitive, namely a SOUP function. This allows the system to take larger, more goal-directed steps through the search space, but it means that many additions and updates to the system's expertise will be done by altering existing functions, rather than by replacing old ones or creating new ones. Yet experts don't always speak in programs. They may watch a run of the system and say, "At that point, such an action should have been taken."

The planning algorithm of the NOAH system maps general knowledge in SOUP code form into knowledge related to a specific problem in the procedural net form. An appropriate algorithm can be designed to perform an inverse mapping from a problem-specific set of actions into a more general SOUP function. If an expert can tell the system how a specific procedural net should be altered for a specific problem, then this algorithm can be used to update the original SOUP code itself, thus allowing the system to learn incrementally.

E. Execution-Time Conditionals

An important use of the facility to intermix planning and execution would be to allow a plan to include decision points that are to be decided upon at execution time. It is sometimes the case that the alternative plans within a disjunctive split specify a set of preconditions that together span all possibilities. An example from an automatic programming domain might be a disjunction based on whether a particular variable were bound to NIL, to an atom, or to a list. When such a condition is discovered, and the system can determine at execution time which disjunct is true, an appropriate critic could mark the disjunction as an execution-time conditional. No planning to achieve any of the preconditions is necessary since one of them is guaranteed to be true.

For certain domains of expertise, where the user has a wide range of choice of possible actions, it will be desirable to encode execution-time choices explicitly into the procedural semantics of the domain. For example, an interactive consultant on the use of a time-sharing system would offer the user alternative ways to use particular system features, and follow a given alternative to greater detail only if the user selected it.

F. Planning for Information Gathering

A similar extension could allow the procedural semantics to specify a specific execution-time test to be performed. For example, a step in a plan to arrange a meeting might be "See if Noah is in his office. If he is, tell him the meeting is at 10:00. Otherwise, leave him a note." The action will have the effect of notifying Noah about the meeting, regardless of the result of the information-gathering portion of the step. The information gathering portion can be expanded further at planning time. One alternative branch will be chosen at execution time, depending on the information acquired. The appropriate branch can then be expanded at execution time.

G. User Models

The question of how to model a human being who is interacting with a computer system is far too broad to be dealt with here. We will simply make a few observations about the ways in which the structure of the procedural net can help in delimiting the areas of competence of a human user.

A first observation is that there are typically many instances of very similar wedges in a plan that a human would carry out with a computer's assistance. Any task is, at some level, a performance by rote of subtasks. The wedge structure of subtasks in the net should allow a wedge-matching algorithm to determine if a wedge to be executed is sufficiently similar to other wedges that have been successfully executed so that no further instructions are necessary other than, effectively, "execute the wedge." The wedge matcher might also note the

differences between wedges, and remind the user of ways in which this wedge were different.

A second observation is that relatively sophisticated users seldom need to know all the individual steps in order; what they need to fully understand a problem is a report of the changes in the plan wrought by the critics. An example of such a request to a sophisticated user is, "Now install the pump top, and be sure to attach the pump brace while putting in the right rear bolt." Perhaps a measure of a user's sophistication is the complexity of criticisms that he does not need to hear about.

H. The Formal Theory of Constructive Criticism

The constructive critics discussed in Chapter 3 were developed in an ad hoc fashion. No attempt has been made to justify the transformations that they perform, or to enable them to generate all valid transformations. However, it should be possible to define an algebra of plan transformations that map plans represented as partial orderings into other partial orderings. It may be possible to develop a body of formal theory about the ways in which interacting subgoals can be dealt with. Some work along these lines has been done in the area of code optimization, where Aho and Ullman [1] developed a complete set of transformations that preserved equivalence among programs.

I. Applications to Robotics and Automatic Programming

The NOAH system can be easily adapted to control a robot vehicle. Most of the problems of controlling a robot from a high level plan are easier to deal with than the corresponding problems involved in assisting a human apprentice. The plan may

always be carried out to a uniform level of detail. The robot will not misunderstand instructions as a human will do. The planner need only produce a single valid plan; it need not try to model all the valid ways in which a human might accomplish a task. And the ways in which an action taken by a robot may fail are more predictable than the ways in which a human's action may fail.

The use of execution-time tests and information-gathering operations would be particularly important for this application.

It would also be relatively easy to adapt the NOAH system to automatic programming tasks. Again, the plans (sequences of program steps, in this case) would always be carried out to the level of greatest possible detail. The nonlinear representation would be ideal for efficient coding of the detailed steps. In fact, many optimizing compilers build some sort of nonlinear representation of the program in order to perform their work. The use of a SOUP-like higher level language would make a compiler's job much easier since the nonlinearity would be explicit and need not be guessed at.

VII SUMMARY AND CONCLUSIONS

A. The Key Ideas That Have Been Explored

In this section we shall reiterate some of the important ideas that have been explored in this report.

1. Using Imperative Semantics to Generate Frame-Like Structures

We have employed an existing methodology for describing a complex of interrelated actions to a computer system: the high-level, goal-oriented PLANNER-like language. The semantics of our SOUP language are quite similar to other PLANNER-like languages. The NOAH system interprets the language in an unconventional manner, however. The statements in the language do not cause the execution of code, but rather cause the creation of frame-like nodes in the procedural net. The nodes have many declarative properties that can be individually accessed. The system can reason efficiently about actions because of the ease of analyzing these declarative properties.

2. The Nonlinear Nature of Plans

This work views plans in a different light than previous general purpose problem solving systems. Rather than requiring plans of action to be linear sequences of actions, we view plans as partial orderings of actions with respect to time. We have demonstrated how, by representing all the freedom of ordering that is innate in a developing plan, a computer system can solve certain problems directly

and without backtracking, whereas a linear representation of the plan forces the inefficient use of backtracking.

3. Planning at Many Levels of Abstraction

We have extended the exploration of problem solving by planning at successive levels of detail that was previously studied in a version of GPS [8] and ABSTRIPS[31]. We have found that this technique is particularly well suited to the representation of plans as partial orderings of actions. NOAH progressively expands a plan to a level of greater detail, and then applies constructive critics to partially linearize the new, more detailed plan. The critics operate by taking a global look at the plan that would be too expensive if all the details were to be analyzed at once.

The earlier systems mentioned above threw away the higher level plans once the desired low-level plan was created. But NOAH uses the higher levels to generate higher level instructions to a human user of the system, to aid in monitoring execution of plans, and to plan efficiently to recover from unexpected situations during plan execution.

4. Execution Monitoring and Error Recovery with Hierarchical Plans

For the cooperative execution of a task by man and computer, the hierarchical structure of the procedural net seems extremely useful. It allows the computer to specify actions at a level of detail no greater than what the human requires. Thus, the maximum information can be transferred for a given amount of instruction, and yet the computer system can go into detail whenever it is needed. Such an approach is essential if the human is to find the system easy to live with.

Even for robotics systems in which the instructions must be given at a fixed low level of detail, the hierarchical representation is useful for establishing which facts about the world must be checked to monitor execution, and for intelligent error recovery.

5. Using Abstract Plans to Model Iterative Operations

An approach to the modelling of iterative actions has been developed that allows the iteration to be specified, at an abstract level, by a single action. This allows for efficient development of plans that involve large numbers of iterations.

6. The Importance of Structure

If the work in this report can be summarized in a phrase, it is this: the structure of knowledge about actions within a computer memory is as important as the content of that knowledge.

The structure of the procedural net allows the planner and execution monitor to deal efficiently with a mass of knowledge about action sequences that would have overwhelmed most previous systems. Most of NOAH's understanding of the actions it is reasoning about is derived from two sources. One is the declarative characterization of actions as nodes in the procedural net. The other is the structure of the net itself that characterizes the interrelationships between actions.

Since it has a good representation for these interrelationships, NAOH can take much of the responsibility for integrating individual subplans. Thus, a task

domain can be specified in a highly modular manner. Rather than encoding a single highly integrated expert, the definer of a domain can encode a set of rather decoupled micro-experts, and simply throw them into the SOUP. Because of the staged execution of the SOUP code, with the building of nodes in the net between stages, the system can efficiently integrate the subplans generated by the micro-experts.

B. What Have We Learned?

In 1960, Plans and the Structure of Behavior, by Miller, Galanter, and Pribram [24] was published as a response by three psychologists to the early work in artificial intelligence. It discusses plan generation, execution monitoring, planning in spaces where details are ignored, the integration of subplans into larger plans, and learning as the development of hierarchical plan structures that can be characterized as single actions in a meta-plan. These are all topics that have been discussed in the preceding pages. Indeed, the title of the present work was chosen to emphasize the underlying similarity. Have fifteen years of work in artificial intelligence made no progress? This chapter is an attempt to answer this question with an emphatic "no."

The production of intelligent behavior by the billions of individual neurons in the cerebrum obviously involves a great deal of structure and organization. It is meaningful to discuss the structures and mechanisms underlying intelligence only in terms of higher order functional units, such as Image (the authors' term for an organism's world- and self-knowledge), Plan, and operational test. The specific nature of these functional units, and the ways in which they interrelate, can be quite

132

poorly specified in a verbal presentation of a hypothesized organization of intelligence. Even a small, obviously inadequate number of functional units can interrelate in so many ways that it is very easy for critical inconsistencies to go unnoticed, and critical interrelationships to go unspecified.

The field of artificial intelligence imposes upon itself a discipline that makes it difficult, though demonstrably not impossible, for a theory about some aspect of intelligence to contain such gaps. That discipline is the implementation of the theory in a working computer program. As Raphael suggested in an early Ph.D. thesis in artificial intelligence [28], the production of a computer implementation provides a check that the specifications for a system are complete and consistent; it suggests ways for avoiding the inconsistencies; its behavior provides insights into the underlying system that may not have been apparent from a mere description; and the resulting program provides a measure of the practicality of the underlying ideas and an experimental tool for testing variations in the original specification.

Though Miller et al. provide insight into human cognition, it is no more than insight. Concepts such as "Image," and "meta-plan" have meanings that are insufficiently defined to be useful as components of a theory of intelligence. One product of years of research in artificial intelligence is progress toward the precise definition of terms that have been used for centuries by insightful men to characterize aspects of intelligence. The precise definitions are being formed by example, by the production of running computer programs that embody these insightful ideas.

The fundamental way in which the work reported here marks progress in

dealing with the issues underlying intelligence is in a move from insight toward understanding. Many of the proposals of Miller et al. have been supported by the demonstration that they can produce intelligent behavior in the NOAH system.

While many of their insights have been supported, others have not been dealt with in this report, since the scope of the work is much more narrowly defined. Their suggestions about memory, natural language, and problem representation have not been touched here. Indeed, it seems disappointing that the ideas in their book did not spark more directed research in artificial intelligence to evaluate the issues that were raised. Of course, without the techniques and tools available today it would have been difficult to make great progress in this research. On the other hand, the desire to do this type of research might have motivated the production of these techniques and tools at an earlier date.

The basic point is that artificial intelligence has developed a methodology and a set of tools and techniques that can be applied to the study of the mechanisms of intelligence. The field might well be able to make faster progress in the understanding of these mechanisms if its workers were to pay more attention to the output of insightful people in other disciplines that were concerned with cognition from different points of view. The discipline imposed by computer implementation helps us to debug our own insights. But we would do well to seek inspiration from related work in fields like psychology, linguistics, and philosophy.

One major insight of Miller, Galanter, and Pribram has been refuted by the work reported here. They often cited the need for an incredible organizational complexity to support their ideas. We have shown that this is not necessarily so.

134

The structure that supports the intelligent behavior of NOAH is complex, as any bridge or building is complex. But, as with bridges and buildings, the organizing principles from which the structure is created are relatively simple. The performance of the NOAH system in the Computer-Based Consultant creates for the casual observer a surprising sense of richness. This suggests not that the mechanisms of the NOAH system are very sophisticated, but rather that the mechanisms underlying intelligence may be simpler than we think.

REFERENCES

1. A.V. Aho, and J.D. Ullman, The Theory of Parsing, Translation, and Compiling, vol. 2: Compiling, Prentice-Hall, Englewood Cliffs, NJ, 845-874 (1973).

2. D.G. Bobrow and B. Raphael, "New Programming Languages for Artificial Intelligence," Computing Surveys, Vol. 6, No. 3 (Sept. 1974).

3. R. Bolles and R. Paul, "The Use of Sensory Feedback in a Programmable Assembly System," Stanford Artificial Intelligence Laboratory, Memo AIM-220, Stanford, CA (Oct. 1973).

4. B.G. Buchanan, G. Sutherland and E. Feigenbaum, "Heuristic DENDRAL: A Program for Generating Explanatory Hypotheses in Organic Chemistry," Machine Intelligence, Vol. 4, B. Meltzer and D. Michie, eds., American Elsevier, New York, pp. 209-254 (1969).

5. J.R. Buchanan and D.C. Luckham, "On Automating the Construction of Programs," Stanford Artificial Intelligence Laboratory, Memo AIM-236, Stanford, CA (May 1974).

6. W.L. Chafe, "Language and Consciousness," Language, Vol. 50, No. 1, pp. 111-133 (March 1974).

7. B.D. Deutsch, "The Structure of Task Oriented Dialogs," Proc. IEEE Speech Symposium, Pittsburgh, PA (April 1974).

8. G.W. Ernst and A. Newell, GPS: A Case Study in Generality and Problem Solving (Academic Press, New York, 1969).

9. S.E. Fahlman, "A Planning System for Robot Construction Tasks," Artificial Intelligence, Vol. 5, No. 1, pp. 1-49 (Spring 1974).

10. R.E. Fikes, "Deductive Retrieval Mechanisms for State Description Models," to appear in Proc. Fourth International Conference on Artificial Intelligence, Tblisi, USSR (September 1975).

11. R.E. Fikes, "Monitored Execution of Robot Plans Produced by STRIPS," Proc. IFIP Congress '71, Ljubljana, Yugoslavia (1971).

12. R.E. Fikes, P.E. Hart, and N.J. Nilsson, "Learning and Executing Generalized Robot Plans," Artificial Intelligence, Vol. 3, No. 4, pp. 251-288 (Winter 1972).

13. R.E. Fikes and N.J. Nilsson, "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving," Artificial Intelligence, Vol. 2, No. 3-4, pp. 189-208 (Winter 1971).

136

14. I.P. Goldstein, "Understanding Simple Picture Programs," MIT Artificial Intelligence Laboratory, Tech. Report AI-TR-294, Cambridge, MA (September 1974).

15. C. Green, "The Application of Theorem-Proving to Question-Answering Systems," Doctoral dissertation, Elec. Eng. Dept., Stanford Univ., Stanford, CA, June 1969. Also printed as Stanford Artificial Intelligence Project Memo AI-96 (June 1969).

16. P.E. Hart, "Progress on a Computer-Based Consultant," to appear in Proc. Fourth International Joint Conference on Artificial Intelligence, Tblisi, USSR (September 1975).

17. P.J. Hayes, "A Representation for Robot Plans," to appear in Proc. Fourth International Joint Conference on Artificial Intelligence, Tblisi, USSR (September 1975).

18. C. Hewitt, "Description and Theoretical Analysis (Using Schemata) of PLANNER: A Language for Proving Theorems and Manipulating Models in a Robot," Ph.D. thesis, Dept. of Math, MIT, Cambridge, MA (1972).

19. D.B. Lenat, "Beings: Knowledge as Interacting Experts," to appear in Proc. Fourth International Joint Conference on Artificial Intelligence, Tblisi, USSR (September 1975).

20. J. McCarthy, P.W. Abrahams, D.J. Edwards, T.P. Hart, and M.I. Levin, LISP 1.5 Programmer's Manual, MIT Press, Cambridge, MA (1965).

21. J. McCarthy and P. Hayes, "Some Philosophical Problems from the Standpoint of Artificial Intelligence," Machine Intelligence, Vol. 4, B. Meltzer and D. Michie, eds., American Elsevier, New York, pp. 463-502 (1969).

22. D.V. McDermott and G.J. Sussman, "The CONNIVER Reference Manual," MIT Artificial Intelligence Lab., Memo No. 259, Cambridge, MA (May 1972).

23. D. Michie, On Machine Intelligence, pp. 149-152 (John Wiley & Sons, New York, NY, 1974).

24. G.A. Miller, E. Galanter, and K.H. Pribram, Plans and the Structure of Behavior, (Henry Holt and Company, New York, 1960).

25. M. Minsky, "A Framework for Representing Knowledge," MIT Artificial Intelligence Lab., Memo No. 306, Cambridge, MA (June 1974).

26. N.J. Nilsson, "A Hierarchical Robot Planning and Execution System," SRI Artificial Intelligence Center, Tech. Note 76, Stanford Research Institute, Menlo Park, CA (April 1973).

27. R. Quillian, "The Teachable Language Comprehender: A Simulation Program and Theory of Language," Comm. ACM, Vol. 12, pp. 459-476 (1969).

28. B. Raphael, "SIR: A Computer Program for Semantic Information Retrieval," in M. Minsky, ed., <u>Semantic Information Processing</u>, MIT Press, Cambridge, MA, pp. 33-145 (1968).

29. R. Reboh and E.D. Sacerdoti, "A Preliminary QLISP Manual," SRI Artificial Intelligence Center, Tech. Note 81, Stanford Research Institute, Menlo Park, CA (August 1973).

30. C. Rieger, "The Commonsense Algorithm as a Basis for Computer Models of Human Memory, Inference, Belief and Contextual Language Comprehension," <u>Proc. Workshop on Theoretical Issues in Natural Language Processing</u>, Cambridge, MA, pp. 180-195 (1975).

31. E.D. Sacerdoti, "Planning in a Hierarchy of Abstraction Spaces," <u>Artificial Intelligence</u>, Vol. 5 No. 2, pp. 115-135 (Summer 1974).

32. G.W. Scragg, "Answering Questions about Processes," in D.A. Norman, D.E. Rumelhard, et. al., <u>Explorations in Cognition</u>, pp. 349-375 (W.H. Freeman and Company, San Francisco, 1975).

33. E.H. Shortliffe, "MYCIN: A Rule-Based Computer Program for Advising Physicians Regarding Antimicrobial Therapy Selection," Stanford Artificial Intelligence Laboratory Memo AIM-251, Stanford, CA (October 1974).

34. G.J. Sussman, "The Virtuous Nature of Bugs," <u>Proc. AISB Summer Conference</u>, pp. 224-237 (July 1974).

35. G.J. Sussman, "A Computational Model of Skill Acquisition," MIT Artificial Intelligence Laboratory Tech. Report AI-TR-297, Cambridge, MA (August 1973).

36. G.J. Sussman and D.V. McDermott, "Why CONNIVing is Better than PLANNing," MIT Artificial Intelligence Laboratory, Memo No. 255A, Cambridge, MA (April 1972).

37. G.J. Sussman and T. Winograd, "MICRO-PLANNER Reference Manual," MIT Artificial Intelligence Laboratory, Memo No. 203 (July 1970).

38. A. Tate, "INTERPLAN: A Plan Generation System that Can Deal with Interactions between Goals," Machine Intelligence Research Unit, Memo. MIP-R-109, University of Edinburgh, Edinburgh (December 1974).

39. R. Waldinger, "Achieving Several Goals Simultaneously," SRI Artificial Intelligence Center, Tech. Note 107, Stanford Research Institute, Menlo Park, CA (July 1975).

40. D.H.D. Warren, "WARPLAN: A System for Generating Plans," Department of Computational Logic, Memo No. 76, University of Edinburgh, Edinburgh (June 1974).

138

41. T. Winograd, "Five Lectures on Artificial Intelligence," Stanford Artificial Intelligence Laboratory, Memo No. AIM-246 (September 1974).

42. T. Winograd, "Procedures as a Representation for Data in a Computer Program for Understanding Natural Language," MIT Artificial Intelligence Laboratory, Tech. Report AI-TR-17, Cambridge, MA, 1971. Published as <u>Understanding Natural Language</u> (Academic Press, New York, 1972).

APPENDIX A

THE FIELDS OF A NODE IN THE PROCEDURAL NET

TYPE – Identifies the kind of action that this node represents. One of the following:

GOAL – A simple action to achieve a goal.

PHANTOM – Like a GOAL, but the goal is expected to be already true when encountered.

BUILD – An iterative action to build up a class of objects.

BREAK – An iterative action to operate on each element of a class of objects.

SOME – A choice point in the plan. The action specified is applied to one of a set of possible arguments.

REPLICATE – A special type of action for representing the multiple iterations of a loop.

ANDSPLIT – Not a true action. This represents a forking in the partial ordering of the time sequence.

ANDJOIN – Not a true action. This represents a joining of parallel paths in the time sequence.

ORSPLIT – Not a true action. This represents a choice point in the time sequence. Only one of the following sequences of actions need be carried out.

ORJOIN – Similar to ANDJOIN.

PLANHEAD – A dummy action indicating the beginning of the plan at each level.

QUERY – A textual string that describes the action represented by the node. The string may be presented to the user of the system, requesting him to perform the action.

PATTERN – The particular expression that the action will cause to be asserted in the world model.

BODY – The code that may be evaluated to expand the action to greater detail.

ADDS - A list of all the expressions that are added to the world model (at the current level of detail) by performing the action represented by the node.

DELETES - A list of all the expressions that are deleted from the world model (at the current level of detail) by performing the action represented by the node.

CHILDREN - The set of nodes in the net that, together, represent a more detailed specification of the action represented by the node.

PARENT - The node of which the current node is a child.

PREDECESSOR - The preceding node (or nodes) in the partially ordered time sequence.

SUCCESSOR - The succeeding node (or nodes) in the partially ordered time sequence.

CONTEXT - The QLISP data context in which the body is to be evaluated. The tree structure of the contexts is essentially isomorphic to that of the parent-child links.

1STCHILD - The first of the node's children in time sequence.

LASTCHILD - The last of the node's children in time sequence.

PRECEDING SIBLINGS - A list of the nodes that precede the current node in time sequence and are children of the current node's parent.

PURPOSE - The last child of the node's parent.

DONTASKUSER - A flag to indicate that the subplan beginning with this action has been so severely altered by criticism at levels of greater detail that it would be misleading to give any instructions at this level.

EXEMPLAR - A special field for REPLICATE nodes, that points to the subplan that represents an example of the expansion of the loop that the node represents.

GENERATED - A list of the formal objects that were generated in the course of building this node.

APPENDIX B

SOUP CODE FOR BLOCKS PROBLEMS

```
(CLEAR
  [QLAMBDA
    (CLEARTOP ←X)
                                        (* Take everything off of
                                        $X, unless $X is the TABLE)

    (OR
      (EQ $X (QUOTE TABLE))
      (QPROG
        (←Y)
        (ATTEMPT (PIS (ON ←Y $X))
            THEN (PGOAL (Clear $Y)
                        (CLEARTOP $Y)
                        APPLY
                        (CLEAR))
                   (PRECLUOE ←Z $Y)
                   (PGOAL (Put $Y on top of ←Z)
                        (ON $Y ←Z)
                        APPLY NIL))
                   (POENY (ON $Y $X)))
        (RETURN])

(PUTON
  (QLAMBOA
    (ON ←X ←Y)
                                        (* Clear $X and $Y, then put
                                        $X on $Y)

    (PANO
      (PGOAL (Clear $X)
             (CLEARTOP $X)
             APPLY
             (CLEAR))
      (PGOAL (Clear $Y)
             (CLEARTOP $Y)
             APPLY
             (CLEAR)))
    (PGOAL (Put $X on top of $Y)
           (ON $X $Y)
           APPLY NIL)
    (POENY (CLEARTOP $Y))))
```

```
(STABILIZE
  [QLAMBDA (STABLE ←X)


                                              (* This code is specific for
                                              the example in the text)
                                              (* To stabilize $X, put it
                                              on A or B)
           (POR (PGOAL (Put $X on A)
                        (ON $X A)
                        APPLY
                        (PUTON))
                 (PGOAL (Put $X on B)
                        (ON $X B)
                        APPLY
                        (PUTON])

(PUTABOVE
  (QLAMBDA (ABOVE ←X ←Y)
                                              (* Clear $X and place it on
                                              top of the tower above $Y)

           (MATCHQ ←TOP (TOWERTOP $Y))
           (PGOAL (Clear $X)
                  (CLEARTOP $X)
                  APPLY
                  (CLEAR))
           (PGOAL (Place $X above $Y)
                  (ABOVE $X SY)
                  APPLY
                  (PLACE))
           (PDENY (CLEARTOP $TOP))))

(PLACE
  (QLAMBDA (ABOVE ←X ←Y)

                                              (* Compute the block on top
                                              of the tower above $Y)
                                              (* Put $X on that block)

           (MATCHQ ←TOP (TOWERTOP $Y))
           (PGOAL (Put $X on $TOP)
                  (ON $X $TOP)
                  APPLY
                  (PUTON))))

(TOWERTOP
  [LAMBDA (BASE)
                                              (* Compute the topmost block
                                              above BASE)

      (ATTEMPT (PIS (ON ←TOP (@ BASE)))
          THEN (TOWERTOP $TOP)
       ELSE BASE])

(TAKEOFF
  (QLAMBDA (NOT (ON ←Y ←X))
                                              (* Take $Y off $X)

           (PGOAL (Clear $Y)
                  (CLEARTOP $Y)
                  APPLY
                  (CLEAR))
           (PDENY (ON $Y $X))
           (PGOAL (Put $Y on top of ←Z)
                  (ON $Y ←Z)
                  APPLY NIL)))
```

The value of $ALLFNS for this set of problems is: (PUTON CLEAR STABILIZE PUTABOVE PLACE TAKEOFF).

APPENDIX C

SOUP CODE FOR THE 'KEYS AND BOXES' PROBLEM

```
(INIT
  [LAMBDA NIL                              (* Set up the initial
                                            world model for the
                                            problem)
     (ASSERT (CONTAINS KEY 1 BOX1 (BAG BOX1 BOX2))
             SIZE INDEFINITE)
     (ASSERT (CONTAINS KEY 1 BOX2 (BAG BOX1 BOX2))
             SIZE INDEFINITE)
     (ASSERT (CONTAINS RED 1 DOOR DOOR)
             SIZE INDEFINITE)
     (ASSERT (CONTAINS NOTHING 0 TABLE TABLE)
             SIZE 0])

(GOTO
  (QLAMBDA (AT ROBOT ←LOC)
           (OR (NEQ $LOC (QUOTE OUTSIDE))
               (ATTEMPT (PIS (AT (KEY ←KEY)
                                 DOOR))
                    ELSE (PGOAL (Bring a key to DOOR)
                                (AT (KEY ←KEY)
                                    DOOR)
                                APPLY $ALLFNS TYPE KEY)))
           (PGOAL (Go to $LOC)
                  (AT ROBOT $LOC)
                  APPLY NIL)))

(PICKUP
  (QLAMBDA (HOLDING ←OBJECT)
           (EMPTYHAND)
           (PIS (AT $OBJECT ←LOC))
           (PGOAL (Go to $LOC)
                  (AT ROBOT $LOC)
                  APPLY
                  (GOTO))
           (PGOAL (Pick up $OBJECT)
                  (HOLDING $OBJECT)
                  APPLY NIL)
           (PDENY (AT $OBJECT $LOC))
           (PDENY (EMPTYHANDED ROBOT))))

(PUTDOWN
  (QLAMBDA (AT ←OBJECT
               ←LOC)
           (PIS (AT ROBOT $LOC))
           (CASES (AT $OBJECT $LOC)
                  APPLY $DEMONS)
           (PGOAL (Put down $OBJECT)
                  (AT $OBJECT $LOC)
                  APPLY NIL)
           (PASSERT (EMPTYHANDED ROBOT))
           (PDENY (HOLDING $OBJECT))))
```

```
(TRANSFER
  (QLAMBOA (AT ←OBJECT
            ←LOCATION)                    (* Move a given object
                                           to a given location)
       (PGOAL (Acquire $OBJECT)
              (HOLOING $OBJECT)
              APPLY
              (PICKUP))
       [ATTEMPT (PDENY (@(PIS (AT ROBOT ←X]
       (PGOAL (Go to $LOCATION)
              (AT ROBOT $LOCATION)
              APPLY
              (GOTO))
       (PGOAL (Put down $OBJECT)
              (AT $OBJECT $LOCATION)
              APPLY
              (PUTOOWN))))

(MOVEINSTANCE
  (QLAMBOA (AT (←TYPE
               ←OBJECT)
            ←LOCATION)                    (* Move an object of a
                                           given type to a given
                                           location)
       [CONO
          ((GETP $OBJECT (QUOTE SPECIFICOBJECT))
           (PIS (AT $OBJECT ←CURLOC))
           (PIS (CONTAINS $TYPE ←OEPTH
                         $CURLOC ←PARTNERS)))
          (T (PIS (CONTAINS $TYPE ←OEPTH
                         ←CURLOC
                         ←PARTNERS]
       [CONO
          ((ATOM $PARTNERS)
           (MATCHQ ←OBJECT
                   (GENITEM $TYPE))
           (PASSERT (AT $OBJECT $PARTNERS))
           (PGOAL (Transfer an object $OBJECT from $PARTNERS
                     to $LOCATION)
                  (AT $OBJECT $LOCATION)
                  APPLY
                  (TRANSFER)))
          (T (EMPTYHANO)
           (PBREAK $PARTNERS (Transfer objects
                               from (@(PRANO (COR $PARTNERS)))
                               to $LOCATION)
                  (ITERATE (@(SUB1 (LENGTH $PARTNERS)))
                        ←PLACE
                        (PROGN (MATCHQ ←MEMBER
                                     (GENITEM $TYPE))
                               (PASSERT(AT $MEMBER $PLACE))
                               (PGOAL(Transfer $MEMBER
                                       from $PLACE
                                       to SLOCATION)
                                     (AT $MEMBER $LOCATION)
                                     APPLY
                                     (TRANSFER]
       (PASSERT (AT ($TYPE $OBJECT)
                  $LOCATION)
              APPLY $DEMONS)))
```

```
(MOVESPECIFICOBJECT
  [QLAMBDA (AT ←OBJECT
               ←LOCATION)                          (* If the OBJECT is of a
                                                   known type, invoke
                                                   MOVEINSTANCE)
                                                   (* Otherwise, the more
                                                   general TRANSFER
                                                   function will be
                                                   invoked)
          (PROG [(TYPE (GETP $OBJECT (QUOTE TYPE]
                (COND
                   [TYPE (PUT $OBJECT (QUOTE SPECIFICOBJECT)
                             T)
                         (RETURN (MOVEINSTANCE
                                     (' (AT ((@ TYPE)
                                             $OBJECT)
                                            $LOCATION]
                   ((FAIL])

(EMPTYHAND
  [LAMBDA NIL

          (* Put down the object currently held where it will
          muddle things up the least)


     (ATTEMPT (PIS (EMPTYHANDED ROBOT))
       ELSE (QPROG (←OBJECT
                    ←PLACELIST)
                [ATTEMPT (PIS (HOLDING ←OBJECT))
                   ELSE (MATCHQ ←OBJECT
                                (GENITEM (QUOTE UNKNOWN]
                (MATCHQ ←PLACELIST
                        (ORDERBYINTEGRITY $PLACES))
                [PSOME $PLACELIST (Put down $OBJECT
                                     in one
                                        of (@(PROR $PLACELIST)))
                   (ITERATE $PLACELIST ←P
                            (PROGN (PGOAL (Go to $P)
                                          (AT ROBOT $P)
                                          APPLY
                                          (GOTO))
                                   (PGOAL (Put down $OBJECT)
                                          (AT $OBJECT $P)
                                          APPLY
                                          (PUTDOWN]
                (PDENY (HOLDING $OBJECT))
                (RETURN (PASSERT (EMPTYHANDED ROBOT])
```

```
(COMPUTECONTAINMENT
  [QLAMBDA (AT ←OBJECT
              ←LOC)

          (* Update the model of what is at LOC when OBJECT is
          placed there)

          (MATCHQ ←OBJTYPE
                  (GETP $OBJECT (QUOTE TYPE)))
          (ATTEMPT (PIS (CONTAINS ←LOCTYPE
                                  ←N
                                  $LOC ←PARTNERS))
              THEN (CONO
                    ((NEQ $LOCTYPE $OBJTYPE)
                     [CONO
                      ((ATOM $PARTNERS)
                       (MATCHQQ ←PARTNERS
                                (BAG $PARTNERS]
                      (PDENY(CONTAINS $LOCTYPE $N $LOC $PARTNERS))
                      [OR (EQ $LOCTYPE (QUOTE NOTHING))
                          (PASSERT (CONTAINS $LOCTYPE
                                             (@(ADD1 $N))
                                             $LOC
                                             (BAG $LOC $$PARTNERS]
                      (PASSERT (CONTAINS $OBJTYPE
                                         (@(ADD1 $N))
                                         $LOC
                                         (BAG $LOC $$PARTNERS]))

(COMPUTEINTEGRITY
  [LAMBDA (PLACE)                                (* Compute how "muddled"
                                                 a given PLACE is)
      (OR (GETP PLACE (QUOTE INTEGRITYMEASURE))
          (QPROG (←PROP
                  ←N
                  ←PARTNERS)
              (RETURN (PUT PLACE (QUOTE INTEGRITYMEASURE)
                           (ATTEMPT (PIS (CONTAINS ←PROP
                                                   ←N
                                                   (@ PLACE)
                                                   ←PARTNERS))
                               THEN (CONO
                                     ((NUMBERP $N)
                                      $N)
                                     (T 9999))
                               ELSE 9999])

(ORDERBYINTEGRITY
  [LAMBDA (PLACES)                               (* Order the list PLACES
                                                 by increasing
                                                 "muddledness")
      (PROG (PLACELIST)
          [MAPC PLACES (FUNCTION (LAMBOA (PLACE)
                  (PUT PLACE (QUOTE INTEGRITYMEASURE)
                       NIL]
          (SETQ PLACELIST (SORT(COPY PLACES)
                               (FUNCTION ORDERTOPRESERVEINTEGRITY)))
          [MAPC PLACES (FUNCTION (LAMBOA (PLACE)
                  (CONO
```

```
                            ((EQP (GETP PLACE (QUOTE INTEGRITYMEASURE))
                                  9999)
                              (SETQ PLACELIST (DREMOVE PLACE PLACELIST]
                  (RETURN PLACELIST]))

(ORDERTOPRESERVEINTEGRITY
   [LAMBDA (PLACE1 PLACE2)
      (ILESSP (COMPUTEINTEGRITY PLACE1)
              (COMPUTEINTEGRITY PLACE2])

(GENITEM
   [LAMBDA (TYPE)
      (CAR (GENOBJECT TYPE])
```

The value of $ALLFNS for this problem is: (GOTO MOVEINSTANCE MOVESPECIFICOBJECT TRANSFER PICKUP PUTDOWN). The value of $PLACES is: (OUTSIDE DOOR BOX1 BOX2 TABLE). The value of $DEMONS is: (COMPUTECONTAINMENT).